M. Sarrafzadeh
D. T. Lee

ALGORITHMIC ASPECTS OF VLSI LAYOUT

VISIT…

ALGORITHMIC ASPECTS OF VLSI LAYOUT

LECTURE NOTES SERIES ON COMPUTING
Editor-in-Chief: D T Lee (*Northwestern Univ., USA*)

Vol. 1: Computing in Euclidean Geometry
Eds. D-Z Du & F Hwang

Lecture Notes Series on Computing – Vol. 2

ALGORITHMIC ASPECTS OF VLSI LAYOUT

Edited by

Majid Sarrafzadeh
D T Lee
Department of Electrical Engineering and Computer Science
Northwestern University
USA

World Scientific
Singapore • New Jersey • London • Hong Kong

Published by

World Scientific Publishing Co. Pte. Ltd.

P O Box 128, Farrer Road, Singapore 9128

USA office: Suite 1B, 1060 Main Street, River Edge, NJ 07661

UK office: 73 Lynton Mead, Totteridge, London N20 8DH

ALGORITHMIC ASPECTS OF VLSI LAYOUT

Copyright © 1993 by World Scientific Publishing Co. Pte. Ltd.

All rights reserved. This book, or parts thereof, may not be reproduced in any form or by any means, electronic or mechanical, including photocopying, recording or any information storage and retrieval system now known or to be invented, without written permission from the Publisher.

For photocopying of material in this volume, please pay a copying fee through the Copyright Clearance Center, Inc., 27 Congress Street, Salem, MA 01970, USA.

ISBN 981-02-1488-X

Printed in Singapore by Utopia Press.

PREFACE

Recent advances in very-large-scale integration technology has resulted in powerful computing systems. As a result, systems have become very complex. To produce effective and reliable digital circuits, automation is necessary. Availability of fast and easily implementable algorithms is essential to the discipline.

In this book we will emphasize on physical design aspect of digital design. Since the cost of fabricating a circuit is a fast growing function of the circuit area, circuit layout techniques are developed with an aim to produce layouts with a small area. Also, smaller area implies fewer defects, hence high yield. Other criteria of optimality, for example, length minimization, delay minimization, and via minimization also have to be taken into consideration. In present day systems, delay minimization is becoming more crucial. The goal is to design circuits that are fast while having small area. Indeed, in "aggressive designs", used for example in medical electronics, speed and reliability are two of the main objectives.

In the past two decades, research has been directed toward automation of layout process. Many invaluable techniques have been proposed. Hierarchical approaches to physical design, being the commonly employed paradigm, solve the layout problem in various stages. We need to optimize each stage while making the problem manageable for subsequent stages. Typically, the following subproblems are considered:

- **Partitioning:** It refers to the task of dividing a circuit into smaller parts. The objective is to partition the circuit into parts such that the size of the components are within prescribed ranges, and the number of connections between the components is minimized. Different partitionings correspond to different circuit implementation. Therefore, a good partitioning can significantly improve circuit performance and reduce layout costs.
- **Floorplanning:** Given a circuit represented by a hypergraph, the floorplanning problem is to determine the approximate location of each module in a rectangular chip area. An important step in floorplanning is to decide the relative location of each module.
- **Placement:** In placement problem, each module is fixed, which means it has fixed shape and fixed terminals. The goal is to decide the "best" position for each module. Usually, some modules have fixed positions, e.g., I/O pads. Though area is what we are concerned with, it is hard to control it. Thus, alternative cost functions are employed. There are two prevalent cost functions: Wire length based and cut based.
- **Global Routing:** The purpose of a global router is to decompose a large routing problem into small, manageable problems (detailed routing). This decomposition is carried out by finding a "rough" path (i.e., sequence of "sub-

regions" it passes) for each net in order to reduce chip size, shorten wire length, and evenly distributed the congestion over routing area.

- **Detailed Routing:** The two-stage global routing followed by detailed routing is an effective technique for realizing interconnections in VLSI circuits. In its global stage, the method first partitions the routing region into a collection of disjoint rectilinear subregions. Typically, the routing region is decomposed into a collection of rectangles. The traditional model of detailed routing is the two-layer Manhattan model with reserved layer where horizontal wires are routed on one layer and vertical wires are routed in the other layer. For integrated circuits, the horizontal segments are typically realized in metal while the vertical segments are realized in polysilicon. In order to interconnect a horizontal and vertical segment, a contact (via) must be placed at the intersection point. Subsequently, the unpreserved layer model was also studied in various literature where both vertical and horizontal wires can run in both layers.

- **Layout Optimization:** In this stage a layout is optimized by, e.g., minimizing the number of vias and compacting the area.

- **Layout Verification:** The layout is tested to satisfy design and layout rules. Also, in more recent CAD packages, the layout is verified in terms of timing and delay.

There are 14 articles presented in this book that deal with various stages of the VLSI layout problem. The first four articles are survey type articles that unify and characterize solutions to various problems. The other articles are research-type.

The first article, "Issues in Timing Driven Layout" by Marek-Sadowska, reviews recent techniques in timing driven placement. Basic timing models are discussed. Recent solutions to the problem have been classified into several categories and their results are compared with techniques that do not address timing issues. Next, Sriram and Kang formulated the placement and global routing problem as a binary-valued optimization problem. In their article "Binary Formulations for Placement and Routing Problems" effectiveness of the formulation is discussed and verified on industrial examples.

An overview of parallel placement techniques is presented by Banarjee in "A Survey of Parallel Algorithms for Placement". Six different classes of algorithms are studied on both shared memory and distributed memory multiprocessors. Makedon and Tragoudas review the circuit partitioning problem in the article "Near Optimal Fast Solution to Graph and Hypergraph Partitioning". Various extensions of a multicommodity flow technique, proposed by Leighton and Rao for circuit partitioning, are surveyed. Steps for improving the time complexity of these algorithms are reviewed.

The first research article is presented by Lengauer and Lugering entitled "LP Formulation of Global Routing and Placement". An overview of integer program-

ming formulation of the global routing problem is presented and a new method for solving it is given. Also, integer programming formulation that integrates placement and global routing is presented. Asano and Tokuyama in their article "Circuit Partitioning Algorithms Based on Geometry Model" investigate a new concept in circuit partitioning. Whereas classical partitioning algorithms are based on graphs, their approach views the problem in a geometric domain. The vertices of the graph to be partitioned is mapped onto the plane, with distance between two points being proportional to the edge-weight between the corresponding vertices. Then, exploiting geometric nature of the transformed problem, an effective partitioning scheme is presented.

The next four articles deal with the various routing models. "Three Dimensional Channel Routing Problem" by Brady, Brown and McGuiness looks beyond two-dimensional routing models and studies the three-dimensional version of the problem. Appropriate models are discussed and near-optimal solutions are presented. Zhou and Preparata consider two commonly employed routing methods in "On the Manhattan and Knock-Knee Routing Modes". Previous lower-bounds on the channel width in the Manhattan model were either based on the notion of density or flux. Here, the authors establish a new lower bound for the channel routing problem that is based on a function of both density and flux. Lower bounds for *channel junction* problems are presented, both in the Manhattan model and the knock-knee model.

The overlap routing model is studied by Gonzalez, Kurki-Gowdara, and Zheng in "Switch-Box Routing under the Two-overlap Wiring Model". In their model, at most two wires are allowed to overlap. A wiring scheme, for assigning wires to conducting layers, is also presented. Finally, routing around two-rectangles is studied by Gonzalez and Lee in "Routing Around Two Rectangles to Minimize the Layout Area". They present a linear time algorithm for routing a set of two-terminal nets around two vertically-aligned rectangles. They prove the produced area is at most two times the area of an optimal routing.

Floorplanning problem is studied in the next two articles. First, Wang and Wong present "A Note on the Complexity of Stockmeyer's Floorplan Optimization Technique". The techniques of Stockmeyer, originally proposed for sliceable floorplans, are extended to consider more general hierarchical floorplans. It is shown that for such class of floorplans the problem becomes difficult. That is, the number of non-redundant implementations of any such floorplan is exponential in the total number of modules. In rectangular dual approach to floorplanning, in order to ensure the existence of a floorplan, the input graph should not have any complex triangles. Tsukiyama and Koike show how to eliminate them in "An Algorithm to Eliminate All Complex Triangles in a Maximal Planar Graph for Use in VLSI Floor-Plan".

Shi considers a fundamental post-processing problem, that is, constrained via minimization. He presents a novel technique in "Constrained Via Minimization and Signed Hypergraph Partitioning". He shows the via minimization problem is

equivalent to a class of graph partitioning problem.

Viewing a circuit as a graph, and embedding the graph instead of the original circuit, provide new insights into the circuit layout problem. Takashaki and Kajitani in their article, "The Virtual Height of a Straight Line Embedding of a Plane Graph", present a graph embedding algorithm. A linear time algorithm for bounding the height of the resulting embedding is presented.

We hope that the collection of articles will not only serve as a resource for researchers wanting to learn most recent results and problems in VLSI layout, but also provide a guide to several classes of open problems in various key areas of VLSI physical design.

Majid Sarrafzadeh and D. T. Lee

CONTENTS

Issues in Timing Driven Layout 1
M. Marek-Sadowska

Binary Formulations for Placement and Routing Problems 25
S. M. Kang and M. Sriram

A Survey of Parallel Algorithms for VLSI Cell Placement 69
P. Banerjee

Approximate Solutions for Graph and Hypergraph Partitioning 133
F. Makedon and S. Tragoudas

Integer Program Formulations of Global Routing and Placement Problems 167
T. Lengauer and M. Lügering

Circuit Partitioning Algorithms Based on Geometry Model 199
T. Asano and T. Tokuyama

The Three-Dimensional Channel Routing Problem 213
M. L. Brady, D. J. Brown and P. J. McGuiness

On the Manhattan and Knock-Knee Routing Models 245
D. Zhou and F. P. Preparata

Switch-Box Routing Under the Two-Overlap Wiring Model 265
T. F. Gonzalez, S. Kurki-Gowdara and S.-Q. Zheng

A Note on the Complexity of Stockmeyer's Floorplan Optimization Technique 309
T.-C. Wang and D. F. Wong

An Algorithm to Eliminate All Complex Triangles in a Maximal Planar Graph for Use in VLSI Floorplan 321
S. Tsukiyama, K. Koike and I. Shirakawa

Constrained Via Minimization and Signed Hypergraph Partitioning 337
C.-J. Shi

The Virtual Dimensions of a Straight Line Embedding of a Plane Graph 357
T. Takahashi and Y. Kajitani

Routing Around Two Rectangles to Minimize the Layout Area 365
T. F. Gonzalez and S. L. Lee

Algorithmic Aspects of VLSI Layout
Eds. M. Sarrafzadeh and D. T. Lee
© World Scientific Publishing Co.

ISSUES IN TIMING DRIVEN LAYOUT

MALGORZATA MAREK-SADOWSKA
Electrical and Computer Engineering Department, University of California
Santa Barbara, CA 93106

ABSTRACT

With the increase of circuit sizes and decrease of feature sizes it is expected that the overall performance of VLSI circuits will be more and more affected by signal propagation efficiency along interconnects. Wires can no longer be modeled as perfect conductors. Some electrical effects, previously considered as secondary, like for example parasitic loading introduced by wires, have to be taken into account at the layout design step. In this paper we address the problems of incorporating timing constraints into the placement and routing of integrated circuits. First, we introduce the timing models, formulate the problem and discuss graph models suitable for its analysis. Next, we give an overview of algorithms resulting in physical designs of improved performance in comparison to those whose objective is just the minimal layout.

Keywords: Layout, placement, timing driven layout, timing driven placement.

1. Introduction

A continuing trend in the design of VLSI chips is the increase of circuit size accompanied by decrease of feature sizes. At the same time, the need for performance based designs continually increases. These two tendencies pose additional problems for computer aided design. A chip which is correctly designed from the circuit and logical standpoint and which has geometrically optimal layout may not function correctly due to various timing problems. It is quite possible that the timing problems of an otherwise correct design are caused solely by the layout of interconnections. In submicron technology, the overall performance of VLSI circuits will be dominated by signal propagation efficiency along the wires on the chip. It is thus crucial to be able to handle the timing problems occurring in the design process of wire dominated chips.

Since the wire delays corrupt designs at the layout stage, the first response was to correct the problem at the physical design level. This has lead to research on timing driven placement, routing and floorplanning, clock tree routing and other issues targeted at layout corrections to decrease the effects of wire caused delays.

But, the problem of wire delays has to be viewed in a broader perspective. Traditionally, the automatic design of integrated circuit has been divided into three basic steps:

1. High level synthesis, which starts with the abstract view of the system and finds a register-transfer structure that realizes the given behavior.
2. Logic synthesis, which converts high-level symbolic descriptions into low-level implementations using logic gates for a given technology. The gate-level description is usually called a netlist.
3. Physical layout which uses the netlist to place gates and route nets to obtain the geometric arrangement of gates and interconnects on a chip.

With designs pushed to performance limits it is very difficult to meet specifications treating each of the design steps separately when decisions made at higher levels affect in an unpredictable way the solution space at the layout level. In particular most of the effort during the high level and logic synthesis is devoted to the minimization of the area occupied by the active devices. This may lead to layouts with excessive wiring. Therefore, it is possible that the flexibility margin left for the layout tools to explore might be too narrow to compensate the wiring delays. It is expected that in a long run a new design methodology will emerge which will allow for controlling the wiring very early in the process. But as long as netlists to be laid out in silicon are not perfect grid graphs, there is a need to improve the performance of a chip by a timing driven placement and routing.

This paper is organized as follows:

First, we discuss the nature and the origin of timing constraints. Next, timing models for modules and wires suitable for physical design are presented. Then, we formulate the timing driven layout problem and finally discuss methods proposed to solve it.

2. Timing Constraints

For the purpose of layout design, a synchronous system can be viewed as a collection of blocks (modules) and nets interconnecting them. We will distinguish four types of blocks, namely: combinational, synchronizing (storage, registers), primary inputs (PIs) and primary outputs (POs). The signals in the system originate in the PIs or synchronizing elements, travel through the combinational blocks and wires interconnecting them and terminate at the POs or synchronizing elements. There is a clock signal which synchronizes data transfers. It is distributed to all the synchronizing elements. Both clock and data signals experience delays between their sources and destinations. The data signals are delayed by the combinational blocks delays and by the delays introduced by the wires. The clock signal suffers from the wire and buffer delays, and in the case of multiple phase clocks, also from the delays introduced by the clock generation circuits.

Performance of a digital system is measured by its cycle time. Shorter cycle time means higher performance. At the layout level, performance of a system is affected by two factors, which are: signal propagation time and clock skew. In case of a single clock signal, clock skew is defined as the maximum difference of the delays from the clock source to the clock pins on latches. For discussion on clock skews for

multiple clock signals, please see [3]. The clock cycle period T, clock skew s, worst case data path delay $d_{\max}$ and offset constant T_0 satisfy the following condition for correct timing

$$T = s + d_{\max} + T_0 \tag{1}$$

The constant T_0 lumps all timing constraints which cannot be changed during the layout process. For detailed description of phenomena considered when setting Eq. (1), please refer to [1]. Equation (1) is frequently referred to as a "long path constraint" because for a given clock period and skew it sets an upper bound on the longest path delay in the system. For a system to function properly there are also requirements imposed on the shortest delays between the storage blocks and/or PIs/POs. They cannot be arbitrarily short. However, the short paths constraints are usually accounted for at the circuit design level. Simply the gates realizing the circuit are chosen such that the delay the signal experiences when traveling through them, even if wires had no delays, would suffice to fulfill the short paths delay requirements.

In order to shorten the cycle time we try to minimize the skew s and the longest path delay $d_{\max}$. When circuit elements are being placed and interconnected only the wire delays can change. Changes in the topology and layout of the clock tree affect the clock skew. Changes in the positions of blocks and the routing of nets change the signal delays in the wires, and in effect the signal paths delays change.

In this paper we will focus on the long path delay and discuss approaches proposed to minimize it at the layout stage. Even though there are several papers published discussing the clock skew minimization problem, we will not address this problem here. We just mention only that the clock routing is usually abstracted as a tree length controlling problem. We feel, however, that the clock tree design involves too many practical issues that make this abstraction somewhat too crude.

3. Timing Models

Before formulating the problem precisely, we have to introduce the basic models and assumptions. To guarantee that the problem can be addressed at the layout step, we need to make the following assumptions regarding the subject circuit:

- The circuit has no asynchronous feedback.
- It is possible to associate explicit clock times with each storage element.

To make the further discussion simpler, we make also one more assumption:

- All circuit gates are modeled as simple delay elements with multiple inputs and single outputs.

The last assumption says that each module has exactly one output pin and one or more input pins. Signal delays between different inputs of the same module and its output are all the same. This assumption is not essential: most of the algorithms accept different pin to pin delays.

Each net in the circuit connects one PI/storage or one output pin of a module with one or more input pins on the modules or with precisely one PO/storage element. A module whose output pins or PI/storage elements have direct connections to input pins of a given module are referred to as fanin of that module. A module whose input pins are connected directly to an output pin of a module are called fanout of that module. If there is a signal path from module A to module B, we say that A is in the transitive fanin of B and B is in the transitive fanout of A.

For the purpose of timing driven layout we assume simple models for module and wire delays. Electrical behavior of a cell is approximated by an equivalent resistance model. It is shown in Fig. 1. R_{out} denotes the output resistance, C_{out} is the output capacitance and C_{in} is the input capacitance of an input pin. The switch is controlled by the logic function of the inputs. In addition to these electrical parameters, a cell is characterized by its area and intrinsic delay T_d.

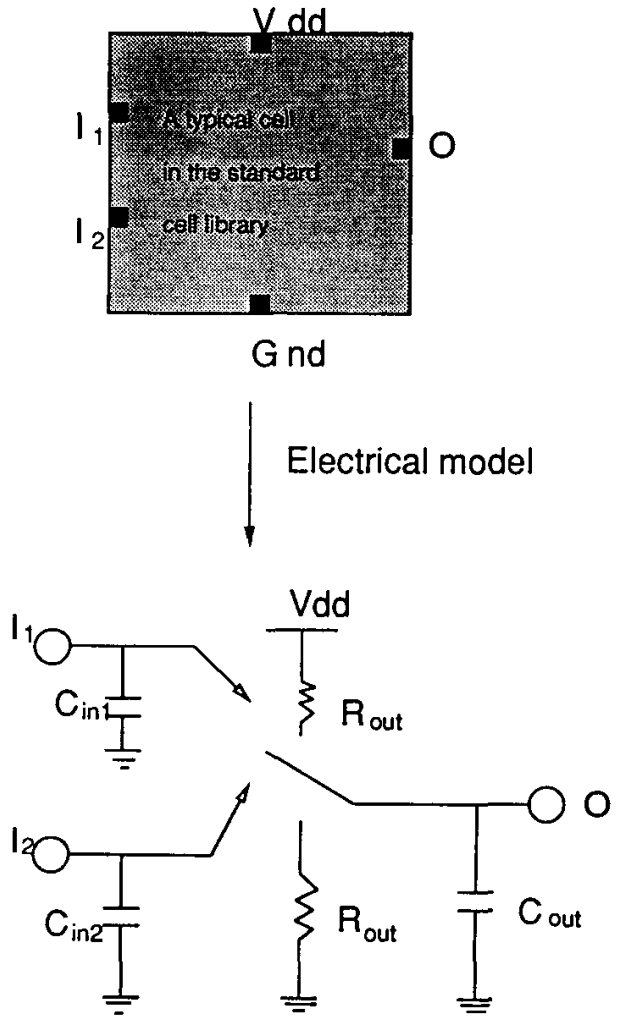


Fig. 1. Electrical model of a cell.

The time delay of a cell is estimated by the lumped RC model depicted in Fig. 2. In the figure, the delay is equal to the product of the output resistance and the capacitive load of the output pin which includes the output capacitance of the cell and the fanout input capacitances. Therefore, referring to the Fig. 2, the delay of a cell A driving the cells B, C and D is expressed by the following equation:

$$D_A = R^A_{\text{out}}(C^A_{\text{out}} + C^B_{\text{in}_2} + C^C_{\text{in}_1} + C^D_{\text{in}_1}) \tag{2}$$

When greater accuracy in delay calculation is required, more precise cell models can be used. In particular, falling and rising delays of a cell should be distinguished as well as different pin to pin delays. Cell delay calculation does not change the timing driven placement algorithms, but it affects their run times.

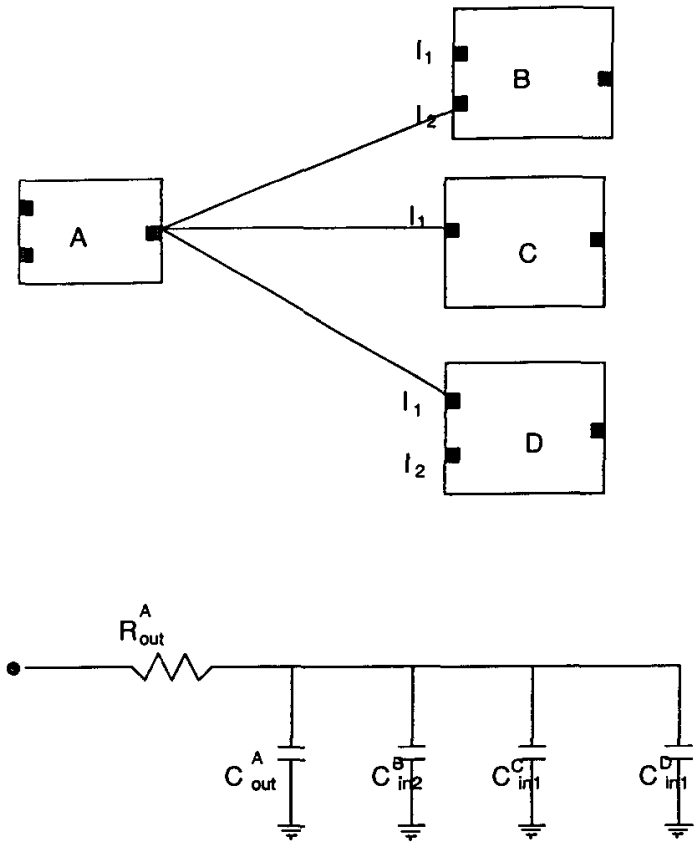


Fig. 2. Delay model of a cell.

Before placement is performed, the topologies and dimensions of particular wires are unknown, therefore the delay model for nets cannot be very accurate. It is usually assumed that a net is modeled by a lumped capacitance. Let C_H and C_V denote the horizontal and vertical capacitance per unit length of the horizontal and vertical interconnect wires respectively. Let R^i_{out} denotes the output resistance of the module m_i whose output drives the net n_i. The delay of the net n_i is determined from the following equation:

$$d_{n_i} = R^i_{\text{out}}[C_H(x_i^{\max} - x_i^{\min}) + C_V(y_i^{\max} - y_i^{\min})] \tag{3}$$

where $x_i^{\max}$, $x_i^{\min}$, $y_i^{\max}$ and $y_i^{\min}$ are the extents of the bounding box of a net n_i.

Frequently, Eq. (3) is simplified further by assuming C_H and C_V being equal. This leads to the following expression for the net delay:

$$d_{n_i} = K_i[(x_i^{\max} - x_i^{\min}) + (y_i^{\max} - y_i^{\min})] \tag{4}$$

The constant K_i is referred to as an inverse of a driving force of a module i. While this model of interconnect delay may introduce some inaccuracy, it is usually sufficient for those designs where the assumption that wires introduce only capacitive delays holds.

When resistive behavior of interconnects cannot be neglected, we may need a wire delay model which accounts for it. Again, during placement the RC tree model for interconnect delay [27] cannot be used because the structures and dimensions of trees are unknown at this time. Usually, in such a case delays are calculated not for entire nets as is the case of Eq. (3), but a delay of each source-sink path is estimated separately. Let m_i be the module driving the net n_i and let m_j be the fanout module of m_i. The delay introduced by wiring between the output pin of m_i and the input pin of m_j can be approximated by the following relationship:

$$d_{ij} = a_{ij} l_{ij}^2 + b_{ij} l_{ij} + c_{ij} \tag{5}$$

where a_{ij}, b_{ij}, c_{ij}, ≥ 0 are constants derived from wire resistance, capacitance and internal module delay. l_{ij} is the Euclidean distance between the pins i and j.

Finally, during the global routing phase the topologies and dimensions of interconnects are decided. More information is therefore available. A net is usually routed as a rooted tree with the driving pin at its root and fanout pins at its leaves. Delays at the nodes of the tree can be calculated and accumulated to obtain the delays at the sinks using the following formulae [27]:

$$d^k_{n_i} = d^j_{n_i} + a * r * c\, l^2_{kj} + b * r\, l_{kj} C^+_{n_i}(j) \tag{6}$$

$$d^{\text{source}}_{n_i} = b\, R^i_{\text{out}} C^i_{\text{out}} \tag{7}$$

where:

r, c are wire resistance and capacitance per unit length, respectively,
$C^+_{n_i}(j)$ is lumped capacitance of all interconnections and loading capacitances connected from node j toward sink terminals,
$d^j_{n_i}$ is delay time from source terminal of the signal to the node j in the tree network. Nodes k and j are connected by a wire segment and the direction of the signal is from the node k to the node j. Again, k and j refer to the nodes in the same tree structure.

4. Problem Formulation

For the purpose of placement and routing, an IC is usually thought of as a collection of modules and nets interconnecting them. Let $M = \{m_1, m_2, \ldots,$

$m_m\}$ represent the modules, $N = \{n_1, n_2, \ldots, n_n\}$ denote the nets and $P = \{p_1, p_2, \ldots, p_p\}$ represent the pins. Customarily, when timing is not addressed during the layout design, goodness of a layout is expressed by some measure of interconnect lengths. Certain measures deal only with 2-pin nets, some can handle multi-pin connections. For 2- pin nets or multi-pin nets decomposed into 2- pin connections the typical measures are:

* Summation of square Euclidean distances:

$$L_E = \sum_N \sum_{p_i, p_j \in N} ((x_{p_i} - x_{p_j})^2 + (y_{p_i} - y_{p_j})^2) \tag{8}$$

*Summation of Manhattan distances:

$$L_M = \sum_N \sum_{p_i, p_j \in N} (|x_{p_i} - x_{p_j}| + |y_{p_i} - y_{p_j}|) \tag{9}$$

Another, frequently used measure for wiring, which does not require net decomposition is:

* Summation of net bounding boxes:

$$L_{BB} = \sum_N \sum_{i \in N} (|x_i^{\max} - x_i^{\min}| + |y_i^{\max} - y_i^{\min}|) \tag{10}$$

When timing driven placement or routing are concerned, the primary goal becomes either the minimization of the longest signal path delay through the system or minimization of one of the above measures (8), (9) or (10) provided that the longest path delay does not exceed the prespecified value.

Let us consider a circuit described by its logic diagram. First we create a graph G. Figure 3 illustrates an example. In the graph G nodes correspond to modules, there is a directed edge from a module m_i to a module m_j if the output pin of the module m_i is connected to an input pin of the module m_j. Each node in G has an associated delay. To simplify the discussion, we assume that wire delays are calculated using the formula (4). In such a case, since each net is driven by exactly one module and a module has one output, the delay behavior of the circuit can be modeled by assigning delays only to the nodes of G. (Note, that using more complex wire delay expression could be modeled by G with wire delays associated with the edges in G and cell delays assigned to the nodes of G.) The delay of each node consists of two parts: one is module delay, which is of known value and the second one is a delay of a net driven by the module. This part of delay depends on physical dimensions of the wires realizing the net and is unknown prior to placement. Thus, loosely speaking, the timing driven layout problem is to determine positions of the modules which in turn determine the net delays such that all the PI/storage — PO/storage paths delays do not exceed the prespecified values.
The long path problem says thus the following:

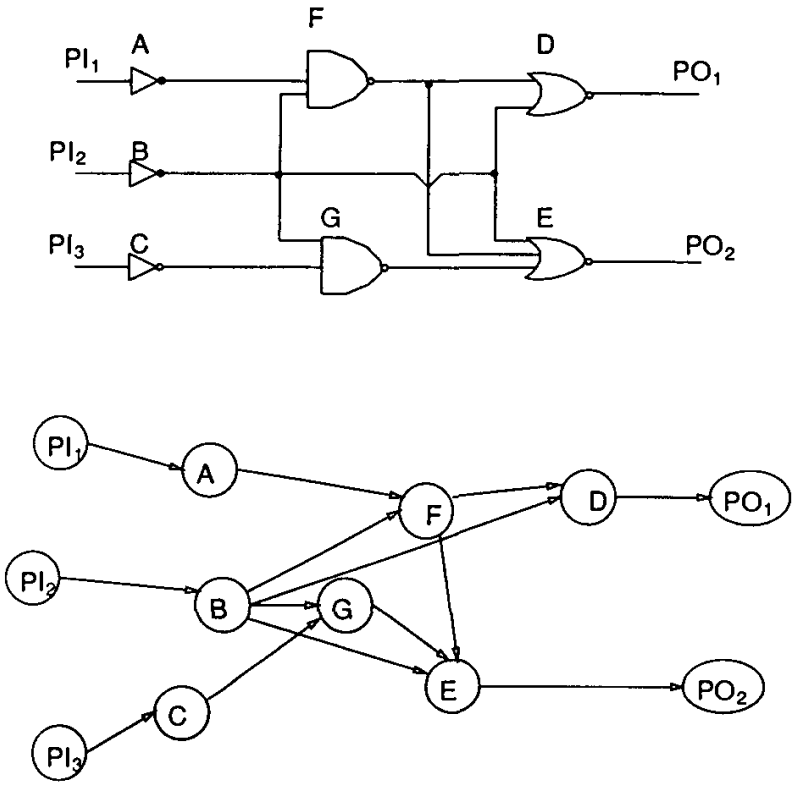


Fig. 3. Circuit diagram and the corresponding graph G.

$$\forall_{k,h}\forall_{P_{kh}} \sum_{i \in P_{kh}} (D_i + d_{n_i}) \leq= H_k \tag{11}$$

where $1 \leq k \leq K$, K is the number of PO/storage elements. H_k is the required arrival time for the primary output O_k; P_{kh} are paths from PI/storage element I_h to PO/storage O_k.

The problem for a timing driven layout tool is to find a placement which fulfills (11) and to perform routing in such a way that (11) still holds.
With each vertex i in the graph G we associate two times [12]:

* T_R^i — required arrival time,
* T_A^i — actual arrival time.

We calculate the actual and required arrival times by the depth first traversal of the graph G using the following formulae:

$$T_A^i = \mathrm{MAX}_{j \in \text{Fanin of } i}(T_A^j + D_j) + T_{d_i} + d_{n_i} \tag{12a}$$

$$T_R^i = \mathrm{MIN}_{j \in \text{Fanout of } i}(T_R^j - T_{d_j}) - D_{n_i} - d_{n_i} \tag{12b}$$

The actual arrival times of the PIs/storage elements are set to 0 and the required arrival times for the POs/storage elements are set to their desired values.

The actual arrival time is a delay of the slowest path from the PIs/storage to the module m_i. The required arrival time is the latest time the signal has to arrive at the output of module m_i to make it to POs/storage on time.
For each node i we define a slack [12] $S(i)$ to be:

$$S(i) = T_R^i - T_A^i$$

A slack tells us the most that the delay of a node can be increased without violating the long path constraints. However, delay increase (equivalent to slack decrease) in one node may influence slacks in other nodes. Slacks are used by many placement and routing algorithms as a measure of how much freedom there is in adjusting positions of modules. Slacks may be determined repetitively during the placement process taking the current information on wire delays into account. Positive slacks characterize those nodes whose delays can be increased, negative delays occur at the nodes whose delays have to be decreased. When wire delays are associated with the source-sink segments as it is the case of RC model for wires, then the slacks are associated with the edge delays rather than the node delays in the graph G.

5. Incorporating Timing Constraints into Placement Algorithms

5.1. Classification

The constraints imposed on layout by the timing considerations are path oriented while most of the tools can handle net related constraints. Thus the timing driven placement algorithms can be classified on the basis of how they handle timing constraints. We can distinguish the following four categories.

1. Timing constraints are translated into the net length bounds prior to execution of the placement program. The net length bounds remain unchanged during the placement.
2. Timing constraints are mapped into a set of weights which are updated during the execution of the placement program.
3. Linear programming or other mathematical optimization approach is used to handle the path constraints and placement directly.
4. Constructive placement is guided by the timing considerations. There is no notion of global weight calculation or global net length constraints.

Below, we will discuss the basic principle behind each of these methods.

5.2. Net Length Constraints

The long paths constraints in the circuit can be transformed into the upper bounds on the net lengths. Besides these there are also other upper bounds on the net lengths which may be caused by the timing models limitations, wired OR voltage drop constraints and other conditions. These second type of constraints are

usually determined during the timing and electrical simulation of the circuit and their fulfillment is a necessary, but not a sufficient condition for meeting the long path delays specifications. Here, we will not discuss how to determine them, but their existence will be accounted for by assuming that there are prespecified spatial bounds on all or some nets.

With the prespecified required arrival times at the POs/storage modules and known delays caused by the combinational modules we can determine how much wire delays can be tolerated by the particular paths in the circuit. The wire path delays can be distributed among the nets constituting them.

Let us consider the problem of net length constraints calculation under the assumption that wires can be modeled as a lumped capacitance. In this case our graph model has delays and slacks associated only with the nodes. Initially, when the placement is not known, we calculate the slacks assuming all net delays d_{n_i} to be zero. If negative slacks occur, we know that the timing driven problem has no solution unless some modules are replaced. Therefore we will consider only the case when no negative slacks are present. The problem of determining net lengths is equivalent to distributing additional node delays such that all the slacks become zero. Let $P_r = (x_1, x_2, \ldots, x_k)$ be a path in the network. We define slack of the path $S(P_r)$ to be:

$$S(P_r) = T_A^1 - \sum_{1<i\leq k} (D_i + d_{n_i}) - T_R^k$$

The zero-slack algorithm is based on the following lemmas [23]:

L1. For any path $P_r = (x_1, x_2, \ldots, x_k)$ through the circuit, $S(P_r) \geq S(x_j)$, $1 \leq j \leq k$.

L2. For each node x in the graph G, there is some path P_x such that $S(x) = S(P_x)$.

From the lemmas L1 and L2 it follows that we can increase node delay by at most the value of its slack. Therefore, we can design a greedy net length assignment algorithm. We could arbitrarily choose a node x having a positive slack. The wire delay of the net x, d_{n_x} can be set to the slack of x, if $S(x)$ is not larger than the maximum delay the net can tolerate, otherwise it is set to the maximum allowable delay, and slacks are recomputed. The algorithm terminates when no nodes with positive slacks are present. The algorithm generates a legal solution but the delays are distributed ununiformly what usually leads to poor wirability. The results can be improved by adding delays to the nodes in a more incremental way. This can be achieved by selecting for delay increase the node whose positive slack is the least. The slack is then distributed as delays over all components whose slacks get reduced to zero.

In general, the problem of increasing node delays such that all slacks in the network become zero can be modeled as a linear programming problem. [23] discusses details of the formulation.

An alternative approach to the one described in [23] was proposed in [39]. There, bounds on net lengths are related more closely to electrical characteristics of a

circuit. An overview and comparison of different algorithms for computing net length constraints is given in [40].

The approach of calculating the feasible net lengths before placement has several advantages. First, it reduces greatly the time devoted to slack computations. Second, it transforms the timing driven placement problem into a placement problem with net length control. Many, if not most of the conventional placement algorithms can be modified to account for the net lengths constraints. There are some disadvantages, though. The placement algorithms can be modified to handle the net length constraints, but they are not really well suited to honor such constraints. So, if a placement algorithm fails to produce a satisfactory solution, we do not know how to change the net bounds to have a better chances of success. The solution space of feasible net lengths satisfying timing constraints is very large, but calculating net lengths before placement is equivalent to picking just one solution. From the standpoint of the placement algorithm, the chosen net bounds are random; we do not know how to iterate in order to converge to net lengths which would lead to a realizable placement.

5.3. Weights for Timing Driven Placement

Most of the existing conventional placement algorithms were designed to optimize a cost function of the type (8), (9) or (10). Therefore, their performance does not change if weights are assigned to nets and if these weights are derived based on timing considerations. The early weight driven timing algorithms worked in the following loop. First, layout was generated without any knowledge about timing requirements of the design. Timing verification then followed. If some interconnects generated by the layout tools created too much parasitic capacitance for the chip to operate properly, the layout step was repeated. Usually, the failing nets got different weights to reflect their criticality and the layout tools attempted to shorten them. An automatic layout/simulation iterative loop based on such a philosophy was proposed for standard cell design at the Bell Telephone Laboratories [6]. Such an approach could succeed in laying out not very aggressive designs, with rather few timing critical paths present. There always could exist a danger that when one group of nets is shortened it may cause other nets to increase their lengths. This makes it possible for other paths to fail in the next step calling for another repetition of the iteration step, and so on. The feedback loop is too large and there is no early warning that the solution will be unfeasible.

A more sophisticated version of shortening of a failing path in conjunction with a min–cut algorithm has been proposed in [2].

The weight based approach is particularly well suited to work in conjunction with hierarchical placement algorithms. Such algorithms divide the placement problems into smaller subproblems which are treated separately. At each level of hierarchy a better approximation of the smallest rectangle containing all the pins of each net is determined or a better approximation of source — sink connections are found.

These are plugged into the timing analysis of the network to determine the new values of weights to be used at the next level of the hierarchy. The timing analysis and the weight calculations are based on the analysis of the entire network, not its part or hierarchical view. Generally the idea of using weights is based on a principle that relatively large weights should be assigned to nets which should be shorter in the layout and smaller weights to those which can afford being longer. Obviously, assigning one set of weights for the entire placement process would be conceptually similar to determining net lengths bounds prior to placement. Usually, the more weight value correcting passes the algorithm has, the better are the results of the weight controlled timing driven placement.

Timing guided weights were used in conjunction with hierarchical placement algorithms as well as with certain "flat" algorithms, for example with the force directed approach [10] and simulated annealing [8]. Below, we describe several approaches to calculating weights.

5.3.1. Node separators of the critical subgraph

Assume the lumped capacitance net delay model. Let G_{crit} be a subgraph of the graph G induced by the nodes whose slacks do not exceed some chosen value S_{crit}. It can be shown, that the subgraph G_{crit} forms a connected graph. We call G_{crit} a critical subgraph, because the delays in its nodes determine the shortest arrival times at the POs/storage modules. Decreasing delays in the nodes of G_{crit} when accompanied by non-increasing delays in the other nodes lead to shorter arrival times for the whole network. We could attempt to decrease delays in all the nodes of G_{crit} but it is sometimes difficult. Instead, we can decrease delays only at some of its nodes and still achieve shorter times. It is clear, that we need to find a group of nodes through which all the paths from PIs to POs pass. Suppose $\{V_p\}$ is a group of such nodes in G_{crit}. We can split the nodes in $\{V_p\}$ into $\{V_p^1\}$ and $\{V_p^2\}$, such an operation will divide G_{crit} into two subgraphs. If the partition of vertices is such that the vertices in $\{V_p^1\}$ have only incoming edges and those in $\{V_p^2\}$ have only the outgoing edges, then decreasing delays in $\{V_p\}$ by b decrease lengths of all the paths in G_{crit} by b. We call such a $\{V_p\}$ a proper node separator. If $\{V_p\}$ cannot be split in such a way, then some paths are shortened more than b while some paths are shortened just by b.

The proper node separators for G_{crit} can be found as follows. First we build an undirected graph G_{cl} which is a transitive closure of G_{crit}. There is an edge between nodes A and B in G_{cl} if there exists a directed path from A to B or from B to A in G_{crit}. Let G_{cmpl} be a complementary graph of G_{cl} in G. The proper node separators for G_{crit} are those maximal cliques in G_{cmpl} which, when deleted, partition G_{cmpl} into two unconnected parts.

The set of vertices separating the graph G_{crit}, $\{V_p\}$, can be found applying a mincut edge partition on G_{crit} and modifying it into a vertex partition. This approach while faster does not guarantee finding proper separators.

In [22] a min-cut algorithm has been described which uses weights derived from the node separators. Weights are assigned to nets as follows: those which are driven by the modules represented by the nodes in $\{V_p\}$ are assigned high weights, other nets in the graph $\{G_{\text{crit}}\}$ are given lower weights and other nets have the weights of zero. At each level of the partitioning hierarchy the critical subgraph G_{crit} is determined and the weights assigned to the nets are recalculated.

5.3.2. Criticality of delay segments

Another approach to deriving appropriate net weights is by introducing some measure of node (capacitive delay model) or edge (RC delay) criticality. This measure tells us how important it is to keep a particular delay segment short. Criticalities are calculated for the entire circuit and allow us to have a more global view of it.

In [19] where the capacitive model was assumed, criticality for each node i was defined as follows:

$$W_i^c = \frac{R_{\text{out}}^i * C_L}{S(i)/l(i)} \tag{13}$$

where: R_{out}^i denotes the output resistance of the module corresponding to the node i, $S(i)$ is a slack of the node i and $l(i)$ is the length of the longest path in terms of a number of nodes from the source nodes in G to the node i.

A more sophisticated version of the criticality measure appears in [20]. First, the delay sensitivity k for delay segments (nodes or edges in G) with respect to loading capacitance or resistance is introduced. Let $d(i)$ be the delay introduced by a wire i. If wires introduce purely capacitive delay then it is sufficient to consider only node delays in the corresponding graph. In such a case i corresponds to a node in G and the delay sensitivity is defined as a derivative of wire delay with respect to wire capacitance:

$$k(i) = (dd(i)/dC)_{C=C_{\text{load}}^i} \tag{14}$$

If the delay is associated with a wire resistance then in the graph G both nodes and edges have delays. Node delays correspond to fixed cell delays and edge delays represent wire delays which are dependent on wire lengths. Here i corresponds to an edge in G and $k(i)$ is defined as a derivative of the wire delay with respect to the wire resistance:

$$k(i) = (dd(i)/dR)_{R=R_{\text{load}}^i} \tag{15}$$

The derivatives are evaluated at the operating points (loading) of the delays.

Next, the normalized slacks for each delay segment are introduced. Consider a delay segment i with a slack $S(i)$. Let the longest path of delay segments including i and such that each of its member has slack equal to $S(i)$ consists of $N(i)$ elements. The normalized slack at the delay segment i is defined as

$$S^*(i) = S(i)/N(i) \tag{16}$$

Slacks may be normalized in different ways; in [20] a normalization with respect to fanin and with respect to fanout and delay sensitivity is mentioned. Criticality W_i^c of a wire segment is then defined as a function of its sensitivity and normalized slack:

$$W_i^c = c_1 * k(i) + c_2/(S^*(i) + s_0) \tag{17}$$

where c_1 and c_2 are scaling parameters to balance between delay sensitivity and slack, s_0 is an offset constant making the denominator positive.

With this choice of weights the wires which are more timing critical (negative slack) or more sensitive to loading (larger k) are shortened with higher priority.

In [20] the criticality measures of delay segments have been used in combination with the capacitance and resistance constraints. These constraints are derived from slacks based on a set of rules which take into account the network topology and structure of nets and yield in uniform slack distribution along the paths and fanouts. The cost function minimized by the partitioning, placement or wiring program when capacitive wire load is assumed, has the following form:

$$L_T = \sum_i W_i^c \delta_C(C_i^{\text{act}} - C_i^{\text{const}}) \tag{18}$$

where:

C_i^{act} is the estimated actual capacitance introduced by the net n_i,
C_i^{const} is the constraint value set on the net n_i,
$\delta_C(\ )$ is a monotone increasing function penalizing the discrepancy between the actual and desired net capacitance.

In the case of RC constraints, a cost function of a form similar to (18) is used. See [20] for details.

The target values for nets resistances and capacitances can include also information of wirability. Dynamic adjustments of their values during the placement process along with recalculation of delay criticalities leads to very good results. The timing constraints appear to be fairly "weak" in a sense that when a solution exists, there are usually very many of them. Obviously, those which are more easily wirable are at premium. The approach proposed in [20] takes both timing and wirability into account.

5.3.3. Analytically derived weights

The weight calculation schemes described so far were based on heuristic observations. It is rather difficult to determine precisely how much net weighting is sufficient to fulfill timing constraints when a cost function of a type (8), (9) or (10) is used. Too much weight imposed on critical nets may cause other path to fail creating new critical nets. To avoid such pitfalls the heuristic weight driven timing placement algorithms require frequent recalculation of weights during the process of placement.

In [31] the problem of net weight control to achieve placement under the delay constraint is analyzed. It is assumed that an upper bound on each net delay has been determined. It has been shown there that the problem of circuit placement optimization with the cost function of the form (8) under the net based delay constraints is exactly the same as optimizing (8) with the weights equal to the optimum Lagrange multipliers of the constraint problem. The equivalence holds for both capacitive and RC wire delay model. This result allows for efficient weight assignment in conjunction with fast algorithms using optimization techniques [32]. Since exact calculation of Lagrangian multipliers is quite expensive, it has been proposed in [31] how to calculate their fast approximations. The simplification in weight calculation is based on the assumption that a weight increase of a particular net has a minor effect on the lengths of the other nets. Under this assumption it is possible to derive the net weights in a closed form. The result is very interesting since it gives theoretical foundations and explains why some of the heuristic weighting schemes work.

Consider the optimum placement problem without timing constraints with cost function of the form (8):

$$L_E = \sum_N \sum_{p_i,p_j \epsilon N} c_{ij}((x_{p_i} - x_{p_j})^2 + (y_{p_i} - y_{p_j})^2) \tag{19}$$

where c_{ij} represents connectivity between the modules m_i and m_j. Let l'_{ij} denote Euclidean net lengths corresponding to the optimum solution of (19). Suppose that the timing driven placement problem was converted into a placement problem under net lengths constraints and let u_{ij} denote the maximum length of interconnection between modules i and j. Let l^*_{ij} be a solution of the constrained problem, i.e. (19) with the upper bounds on nets being u_{ij}. Denote by a_{ii} the connectivity from module m_i to all adjacent modules:

$$a_{ii} = \sum_k c_{ik} \tag{20}$$

The approximated weight values are given by the following equation:

$$W_{ij} = \frac{(l^*_{ij} - u_{ij})/u_{ij}}{1/a_{ii} + 1/a_{jj} - 2c_{ij}/(a_{ii}a_{jj})} \tag{21}$$

From the Eq. (21) it follows that the value of the weight that needs to be imposed on a net in order to change its length from an optimum in the unconstrained case to the target value in the constrained case depends on:

1. relative size of the change caused by imposing the constraints (numerator of the expression (21)),
2. how strongly the modules m_i and m_j are mutually connected and how strong is their connectivity to the rest of the circuit (denominator of (21)).

The dependence on slacks is included in the target net lengths u_{ij}. Note, that the analytical calculation of the weights does not solve the timing driven placement completely, but is certainly a very important contribution. The authors of [31] do not discuss how the target net lengths should be determined. Following the approach of [23] might not be sufficient since module overlapping and wirability should be taken into account.

5.4. Optimization Methods to Handle Path Constraints

5.4.1. Linear programming formulation

All the approaches discussed so far were based on a principle of converting path oriented constraints into net oriented constraints. At best the net type constraints were dynamically adjusted at each iteration of the placement algorithm.

For the capacitive wire delay model one can couple spatial constraints imposed by placement itself with spatial constraints introduced by the timing into a set of linear inequalities and formulate a linear program [14]. Consider a net n_i. Let η_i, σ_i, ω_i, ε_i denote the top, bottom, left and right coordinates of its bounding box, respectively. For each pin p_j in the net n_i we have the following set of inequalities imposed on its x-y coordinates:

$$\eta_i \geq y_{ij} \tag{22a}$$

$$\sigma_i \leq y_{ij} \tag{22b}$$

$$\varepsilon_i \geq x_{ij} \tag{22c}$$

$$\omega_i \leq x_{ij} \tag{22d}$$

The delay introduced by the net n_i is described by a formula of the form of (3):

$$d_{n_i} = K_V(\varepsilon_i - \omega_i) + K_H(\eta_i - \sigma_i) \tag{23}$$

For input pins of all the modules except PIs a linear inequality describing the latest arrival signal are set as follows:

$$\forall_{j\varepsilon \text{ Fanin of } i} t_A^i \geq (t_A^j + D_j) + T_{d_j} + d_{n_m} \tag{24a}$$

where t_A^i denotes the actual arriving time at the input pin i, j is an input pin of the fanin module of the module to which i belongs. n_m is a net to which the connection ij belongs. For the POs or registers, the equation expressing the timing relationship is slightly different. Their required arrival times r_i are specified. For each endpoint pin an additional variable M is inserted into the timing inequality. It represents the minimum slack for all the cells, so $M \geq \min S(i)$ insures that all path delays at the POs/registers fulfill the required time constraint. The equation for the endpoints are as follows:

$$\forall_{j\varepsilon \text{ Fanin of } i} t_R^i \geq (t_A^j + D_j) + T_{d_j} + d_{n_m} + M \tag{24b}$$

t_R^i denotes the required arrival time at the input pin i of the PO/storage module. The objective function to the linear program is:

$$\max(M - \alpha W) \tag{25}$$

where

$$\alpha = \frac{\kappa \bar{R}}{|N|}$$

here κ is a user specified constant, $\bar{R}$ is the average output cell resistance in the network, and $|N|$ is the number of nets. W is the total network capacitance introduced by the wires and therefore is a measure of the weighted sum of interconnects. The function $(M - \alpha W)$ is minimized subject to (22), (24) with additional constraints centering the cells within the placed region (see [14] for details). The cost function captures both qualities of the layout we wish to achieve: it tries to shorten the longest path delay and to decrease the total wire length. However, since the slot constraints and the module non-overlap conditions are not modeled with a sufficient accuracy, a solution to the above linear problem does not produce the desired results in one iteration. It has to be coupled with some partitioning scheme which at each iterative step limits some/all feasible bounding boxes for nets. The linear program has to be solved after each partition to guide the subsequent one.

The linear programming approach has an advantage of taking into account all the flexibility of the path based constraints. It explores the solution space guided by spatial considerations and is only relatively weakly constrained by timing. It has therefore good chances in succeeding in determining a wirable layout which fulfills timing constraints. The disadvantage of the LP approach is the repetitive need of solving large problem instances. It seems that the practical placement problems lead to LP instances an order of magnitude larger than the linear programming software can reasonably handle.

5.4.2. Nonlinear program formulation

When wire delays exhibit an RC behavior the delays of source-sink wire segments are described by the relationship (5). In such a case an LP formulation is not appropriate due to nonlinear dependence of delay on wire length. In [26] a timing driven placement obtained by solving a nonlinear program was proposed. In order to simplify module non-overlap constraints it was assumed that modules are modeled as circles with areas equal to the original modules. The timing constraints are considered for a subset of critical paths. The objective function is a summation of normalized path wire delays over the critical paths. Constraints are: 1) slacks of critical paths are non-negative, 2) modules do not overlap, 3) modules are placed within the prespecified bounding boxes. The timing driven placement formulated in this form is relatively difficult to solve when the nonlinear program has no regular structure and becomes impractical for large problem instances. Its

main applications are in timing driven floorplanning where the number of blocks is small.

In [36, 37] the timing driven placement for capacitive wire delay is considered. The optimized function is a summation of squared wire lengths (8) such that the long path constraints are fulfilled. The problem is formulated as a nonlinear program but due to the observation that it is sparse and exhibits certain regularity it is possible to solve it very efficiently.

The abstract model used in [36, 37] is as follows. The network is modeled by a graph $\bar{G}$ in which the set of vertices $\bar{V}$ represents the pins, and edges represent signal flow. There is an edge connecting vertices v_i and v_j and directed from i to j if the pin p_i is an output pin connected to an input pin p_j or the pin p_i is an input pin and p_j is an output pin of the same module. Edges representing the signal flow through the modules are referred to as internal edges E_I and the edges corresponding to the flow through the wires are external edges E_E. Internal edges have fixed delays assigned to them representing cell pin to pin delays and delays of external edges can be calculated from a formula (3). With each node we can associate an actual and required arrival times in the same way as we did it previously for the nodes in graph G (12a), (12b). Let w_{cell} be the combined vector of x and y coordinates of the cell positions:

$$w_{\text{cell}} = \begin{bmatrix} x \\ y \end{bmatrix}$$

and w_{time} be the vector of node actual arrival times in $\bar{G}$. Then, the vector

$$w = \begin{bmatrix} w_{\text{cell}} \\ w_{\text{time}} \end{bmatrix}$$

is a $2|m| + |p|$ vector and contains all variables in the problem formulation. Let the cost function be the summation of square wire lengths (8) weighted with net connectivities c_{ij} which represent the number of nets that the modules i and j share. Using the matrix notation, (8) can be stated as follows:

$$L(x, y) = (x^T B x + y^T B y) + c^T x + d^T \tag{26}$$

The vectors c and d are contributions from the fixed modules (external pads) and B is a symmetric matrix, $B = D - C$ where C is connectivity matrix and D is a diagonal matrix with $d_{ii} = \sum_{j=1,n} c_{ij}$. This cost function has to be optimized with respect to the timing constraints. The formula (26) can be rewritten to be a function of w. To do so, the following quantities are introduced:

$$Q = \begin{bmatrix} B & 0 & 0 \\ 0 & B & 0 \\ 0 & 0 & B \end{bmatrix} \tag{27}$$

Q is the $(2|m| + |p|) * (2|m| + |p|)$ matrix.

$$b = \begin{bmatrix} c \\ d \\ 0 \end{bmatrix} \tag{28}$$

b is a $(2|m| + |p|)$ vector. The cost function can be written as

$$L = 1/2(w^T Q w + b^T w) \tag{29a}$$

The optimization problem can be stated as follows:

$$\text{minimize } L \tag{29b}$$

subject to

$$t_A^i \geq t_A^i + d_{ij} \quad \text{for each edge on } \bar{G} \tag{29c}$$
$$t_A^j \leq t_R^j \quad \text{for } j \text{ being POs/storage nodes} \tag{29d}$$

The optimization problem formulated above if attempted to solve without making use of its structure constitutes a complicated numerical problem. However, several properties of (29) can be recognized what in effect leads to substantial simplifications. The facts which form a basis for simplification are [36]:

F1. If there exists at least one fixed module and the graph $\bar{G}$ is connected then $x^T Bx$ and $y^T By$ are positive definite.

F2. Any local minimum of (29a,b) is also a global minimum.

F3. The satisfaction of the Kuhn-Tucker first order optimality conditions are sufficient for a point to be global minimizer of (29a,b).

Let A^* denote the vector function of the active constraints (those which are satisfied with equality) at the global minimum w^* and ∇A^* is the associated Jacobian matrix. If at w^* the ∇A has full rank then from F3 it follows that there exist Lagrangian multipliers λ such that the following hold:

$$\begin{gathered} \nabla L(w^*) + \lambda^T \nabla A^* = 0 \\ \lambda^T \nabla A^* = 0 \\ \lambda \geq 0 \end{gathered} \tag{30}$$

The key fact regarding the structure of the the problem (29) can be stated as follows [36]:

F4. The active timing constraints and delay equations at the optimal solution w^* can be replaced by an equivalent set of active constraints which consists of equation for arcs on paths in $\bar{G}$ from vertices being sources in $\bar{G}$ to vertices being sinks in $\bar{G}$.

The property F4 is of major significance since it allows for a drastic reduction in the size of the active constraints. In [36] it has been proposed to solve (29) using the dual active set algorithm. Making use of the F4 it can be shown that it is possible to solve (29) with the dual algorithm in a finite number of iterations. For details see [36].

As previously, solving the optimization problem (29) once is not sufficient to find a legal placement which besides fulfilling the timing requirement would be additionally wirable and has no overlaps. In [36] it has been proposed to solve the wirability — overlaps problem by placing modules on the slots. To achieve this, (29) is repetitively solved with additional centering constraints. In the first level of the hierarchy we wish to have the median of module positions to coincide with the center of the chip. Next we partition the modules into two groups and set centering constraint for each of the groups. This process continues up to the desired granularity. Note, that the centering constraints do not force modules to physically reside in their partitions what allows for their migration in the course of iterations.

5.5. Performance Driven Constructive Placement

5.5.1. Window Approach [18]

The motivation behind the window concept is to transform the timing information into geometric domain which would allow for casting timing driven placement into a placement problem with preferred positions for some/all blocks. Consider a path which originates in some PI/register and terminates in another PO/register. It is assumed that the memory elements, PIs and POs have been preplaced. The coordinates of the path's terminating points determine the possible smallest enclosing rectangle in which the elements of the path can be placed. This in turn determines the smallest possible delay the path can have: the module delays and load characteristics introduced by the path are known and the best possible situation would be if all the wire delay would be contributed by the module with the smallest multiplicative constants in Eqs. (3) or (5), depending on the assumed model. Placing the modules constituting a path in its window such that as little as possible net zigzagging is introduced yields in short path delay. This is the thrust of the algorithm proposed in [18]. Obviously, to achieve correct timing usually many critical and sub-critical paths have to be considered. Also, the same module may belong to several paths whose initial windows may even not intersect. In such a case the windows are inflated to allow for intersection. The placement algorithm determines optimal window for a group of elements which are placed within it using constructive approach. After placing a group of elements, path delay information is updated and new windows are selected.

It seems that the results that this algorithm can achieve depend very strongly on the pad placement. It seems that the method should work well for pad placement following the signal flow in the network.

5.5.2. Adaptive control approach [33, 34]

Yet another approach is to use the timing constraints to guide a constructive placement. Suppose that a partial placement has been obtained. In the beginning of the process the partial placement might be just the pad placement or a placement of just one element. The partial placement is analyzed and a look ahead strategy

is used to determine the potentially critical paths and based on this information the next element is selected and properly placed. On the partial placement an incremental global routing is executed allowing for very accurate timing information about the timing data regarding the placed portion. In this scheme the priorities on which nets should be kept short keep changing during the course of the placement. The placement algorithm may have some more flexibility if the partial result seen so far has better characteristics than initially expected. On the other hand it allows for somewhat worse partial result in a hope to improve later. The quality of results obtained using this approach depend on the accuracy of critical paths prediction. This is not a trivial problem and solution to this problem was proposed in [34].

The adaptive control placement algorithms performs two major steps:

1. The next module to be placed is determined. The next module is the one for which a function being a linear combination of connectivity and delay is maximized.
2. Position for the selected module is determined. Here the position is chosen such that the linear function of interconnect timing delay and a measure of unevenness of the boundary of the partial placement is minimized.

After executing the above loop, the timing information is incrementally updated and the parameters describing the quality of the present result along with the future predictions are modified accordingly.

This approach has an advantage of decreasing the size of the problem both in the timing and geometric domain in its due course. It is very flexible and allows to experiment with various look ahead and cost criteria. It seems that it could be used as a detailed timing driven placer following initial placements obtained by other algorithms. To date there are no data available on comparisons on the same examples between hierarchical and constructive timing driven placement. The comparisons of placement data for examples without timing constraints suggest that iterative algorithms are capable of obtaining better results. It would be interesting to see what is the effect of timing constraints on the quality of results for different types of algorithms.

6. Conclusions

In this paper we have discussed the problem of timing driven placement. We have presented the timing models suitable to capture the relationship between geometric and timing domains. We gave an overview of the representative methods developed for obtaining placements under the timing constraints. Our feeling is that the problem of incorporating timing into the physical design should be addressed at the higher levels of abstraction. The structure of the circuit to be laid out should have certain regularity allowing for easy predictions of wire delays. Also, there should exist mechanisms allowing for tradeoffs between active device and wire area driven by such predictions. It seems like designing layouts of unstructured circuits under timing constraints is still fairly unpredictable.

7. Acknowledgements

The author acknowledges support from the National Science Foundation through the grant MIP 88-03711.

References

1. H. B. Bakoglu, "Circuits, interconnections and packaging for VLSI", Addison–Wesley Publishing Company, 1990.
2. M. Burstein and M. N. Youssef, "Timing influenced layout design", IBM Research Report, RC 10862 (#48565), 11/15/84, see also *Proc. 22nd Design Autom. Conf.*, 1985, pp. 124–130.
3. A. F. Champernowne, L. M. Bushard, J. T. Rusterholz and J. R. Schomburg, "Latch-to-latch timing rules", *IEEE Trans. on Comp.*, Vol. 39, No. 6, June 1990, pp. 798–808.
4. A. H. Chao, E. M. Nequist and T. D. Vuong, "Direct solution of performance constraints during placement", *Proc. Custom Integrated Circ. Conf.*, 1990, pp. 27.2.1–27.2.4.
5. W. E. Donath, R. J. Norman, B. K. Agrawal, S. E. Bello, S. Y. Han, J. M. Kurtzberg, P. Lowy and R. I. McMillan, "Timing driven placement using complete path delays", *Proc. 27th Design Autom. Conf.*, 1990, pp. 84–89.
6. A. E. Dunlop, V. D. Agraval, D. N. Deutsch, M. F. Jukl, P. Kozak and M. Wiesel, "Chip layout optimization using critical path weighting", *Proc. 21st Design Autom. Conf.*, 1984, pp. 133–136.
7. C. M. Fiduccia and R. M. Mattheyses, "A linear time heuristic for improving network partitions", *Proc. of 19th Design Autom. Conf.*, 1982, pp. 241–247.
8. Ph. de Forcrand and H. Zimmermann, "Timing-driven auto-placement", *Proc. Int. Conf. on Comp. Design*, 1987, pp. 518–521.
9. Y. Fujihara, Y. Sekiyama, Y. Ishibashi and M. Yanaka, "DYNAJUST: An efficient automatic routing technique optimizing delay conditions", *Proc. 26th Design Autom. Conf.*, 1989, pp. 791–794.
10. T. Hasegawa, "A new placement algorithm minimizing path delays", *Proc. Int. Conf. on Circ. and Syst.*, Singapore 1991, pp. 2052–2055.
11. P. S. Hauge, R. Nair and E. J. Yoffa, "Circuit placement for predictable performance", *Proc. of Int. Conf. on Comp. Aided Design*, 1987, pp. 88–91.
12. R. B. Hitchcock, Sr., G. L. Smith and D. D. Cheng, "Timing analysis of computer hardware", *IBM J. Res. Develop*, Vol. 26, No. 1, January 1982, pp. 100–105.
13. M. A. B. Jackson, E. S. Kuh and M. Marek-Sadowska, "Timing-driven routing for building block layout", *Proc. of Int. Symp. on Circ. and Syst.*, 1987, pp. 518–519.
14. M. A. B. Jackson and E. S. Kuh, "Performance driven placement of cell based ICs", in *Proc. 26th Design Autom. Conf.*, 1989, pp. 376–381.
15. B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs", *Bell Syst. Tech. J.*, Vol. 49, 1982.
16. E. S. Kuh, A. Srinivasan, M. A. B. Jackson, M. Pedram, Y. Ogawa and M. Marek-Sadowska, "Timing driven layout", *Proc.of the Syn. and Simul. Meeting and Int. Interchange*, SASIMI'90, October 1990, Kyoto, Japan, pp. 263–270.
17. U. Lauther, "A min-cut placement algorithm for general cell assemblies based on graph representation", *Proc.16th Design Autom. Conf.*, 1979, pp. 1–10.
18. I. Lin and D. H. C. Du, "Performance-driven constructive placement", *Proc. 27th Design Autom. Conf.*, 1990, pp. 103–106.

19. S. P. Lin, M. Marek-Sadowska and E. S. Kuh, "Delay and area optimization in standard cell design", *Proc. 27th Design Autom. Conf.*, 1990, pp. 349–352.
20. W. K. Luk, "A fast physical constraint generator for timing driven layout", *28th Design Autom. Conf.*, 1991, pp. 626–631.
21. M. Marek-Sadowska, "Route planner for custom chip design", *Proc. of Int. Conf. on Comp. Aided Design*, 1986, pp. 246–249.
22. M. Marek-Sadowska and S. P. Lin, "Timing driven placement", *Proc. Int. Conf. on Comp. Aided Design*, 1989, pp. 94–97.
23. R. Nair, C. L. Berman, P. S. Hauge and E. J. Yoffa, "Generation of performance constraints for layout", *IEEE Trans. on Comp. Aided Design of Integrated Circ. and Syst.*, Vol. 8, No. 8, August 1989, pp. 860–874.
24. Y. Ogawa, T. Ishi, Y. Shirashi, H. Terai, T. Kozawa, K. Yuyama and K. Chiba, "Efficient placement algorithms optimizing delay for high-speed ECL masterslice LSI's", *Proc 23rd Design Autom. Conf.*, 1986, pp. 404–410.
25. S. Prasitjutrakul and W. Kubitz, "Path-delay constrained floorplanning: A mathematical programming approach for initial placement", *26th Design Autom. Conf.*, 1989, pp. 364–369.
26. S. Prasitjutrakul and W. Kubitz, "A performance-driven global router for custom VLSI chip design", submitted for publication.
27. T. Sakurai, "Approximation of wiring delay in MOSFET LSI", *IEEE J. of Solid-State Circ.*, Vol. 18, No. 4, 1983, pp. 418–426.
28. S. Teig, R. L. Smith and J. Seaton, "Timing driven layout of cell-based IC's", *VLSI Syst. Design*, May 1986, pp. 63–73.
29. H. Terai, F. Goto, K. Wakai, T. Kozawa, M. Edagawa, S. Hososaka and M. Hashimoto, "Basic concepts of timing-oriented design automation for high-performance mainframe computers", *28th Design Autom. Conf.*, 1991, pp. 193–198.
30. M. Terai, K. Takahashi and K. Sato, "A new min-cut placement algorithm for timing assurance layout design meeting net length constraint", *27th Design Autom. Conf.*, pp. 96–102.
31. R. S. Tsay and J. Koehl, "An analytic net weighting approach for performance optimization in circuit placement", *28th Design Autom. Conf.*, 1991, pp. 620–625.
32. R. S. Tsay, E. S. Kuh and C. P. Hsu, "Proud: A sea of gates placement algorithm", *IEEE Design and Test of Comp.*, December 1988, pp. 44–55.
33. S. Sutanthavibul and E. Shragowitz, "An adaptive timing-driven layout for high speed VLSI", *Proc. 27th Design Autom. Conf.*, 1990, pp. 90–95.
34. S. Sutanthavibul and E. Shragowitz, "Dynamic prediction of critical paths and nets for constructive timing-driven placement", *Proc. 28th Design Autom. Conf.*, 1991, pp. 632–635.
35. S. Sutanthavibul, Habib Youssef and E. Shragowitz, "Cell based physical design under timing constraints", *Proc. Int. Symp. on Circ. and Syst.*, 1990, pp. 877–880.
36. A. Srinivasan, "An algorithm for performance-driven initial placement of small cell ICs", *28th Design Autom. Conf.*, pp. 636–639.
37. A. Srinivasan, K. Chaudhary and E. S. Kuh, "Ritual: An algorithm for performance driven placement of cell based ICs", UCSB Memo No.UCB/ERL M91/47, see also *Proc. Int. Conf. on Comp. Aided Design*, 1991, pp. 48–51.
38. H. Youssef, E. Shragowitz and S. Sutanthavibul, "Timing analysis of cell-based VLSI designs", to appear in International Journal of Computer-Aided VLSI Design.
39. H. Youssef and E. Shragowitz, "Timing constraints for correct performance", *Proc. Int. Conf. on Comp. Aided Design*, 1990, pp. 24–27.

40. H. Youssef, R. B. Lin and E. Shragowitz, "Bounds on net delays for physical design of fast circuits", *Proc. Int. Conf. on Very Large Scale Integration*, VLSI'91, pp. 3.b.3.1–3.b.3.7.
41. H. Youssef, E. Shragowitz and L. C. Bening, "Critical path issue in VLSI designs", *Proc. Int. Conf. on Comp. Aided Design*, 1989, pp. 520–523.

Algorithmic Aspects of VLSI Layout
Eds. M. Sarrafzadeh and D. T. Lee
© World Scientific Publishing Co.

BINARY FORMULATIONS FOR PLACEMENT AND ROUTING PROBLEMS

S. M. KANG
214 Coordinated Science Laboratory, University of Illinois
1308 W. Main Street, Urbana, Illinois 61801, USA

and

M. SRIRAM
4303 Beckman Institute, University of Illinois
405 N. Mathews Ave., Urbana, Illinois 61801, USA

ABSTRACT

The module placement and global routing problems are formulated as binary-valued optimization problems. The binary formulations allow these complex problems to be solved using specialized hardware or efficient mathematical software. The placement problem has been formulated as a hierarchical quadratic binary optimization problem, which is of a form that can be solved using a new neural network model. The global routing problem, with particular emphasis on the Sea-of-Gates and Custom Logic array design styles, has been formulated as a zero-one integer linear program, and solved using efficient zero-one optimization methods. Both of these approaches have produced high-quality results on industrial circuits.

Keywords: Placement, routing, neural networks, linear programming, optimization.

1. Introduction

The development effort of VLSI chips increases superlinearly as the number of on-chip circuit functions increases. Without innovative design methodologies and advanced computer aids, the design time, starting from chip architecture, logic and circuit design, to mask generation, can even be longer than the useful life of one technology generation. In order to take full advantage of state-of-the-art technologies, the design cycle time should be long enough to allow important design optimizations, and at the same time short enough to allow technology maturing for satisfactory chip production yield. Long design cycles allow more time for design optimizations, but may miss the critical market window for the chip or system product. The physical design of VLSI chips involves a broad range of design activities including the complex layout, chip timing analysis, and design optimization for both speed performance and production yield. Computer aided design tools are essential for physical design and many tools have been successfully developed in the past. However, as the design complexity increases steadily, so does the need

for better CAD tools. Perhaps, the most basic design problems lie in the domain of placement and routing. Indeed, placement and routing steps are constantly required at many levels of the physical design hierarchy. This is one of the main motivations pushing for on-going research on this topic.

In this chapter, we will describe binary formulations for placement and routing problems in VLSI design. These formulations can be applied to a variety of different design styles, but they are illustrated here for standard cells and sea-of-gates (SOG) design styles. Systematic formulations enable the use of powerful optimization techniques to obtain good solutions. Section 2 will deal with the binary formulation of the module placement problem and its solution using a modified Hopfield neural network. Section 3 will address 0-1 integer programming approaches for the global routing of sea-of-gates. As demonstrated through test results on practical examples, these binary techniques generate promising results and continued research on this topic may provide more improvements.

2. The Module Placement Problem

Given a set of predesigned modules, which may be standard cells or custom modules, and a netlist decribing the connections to be made between them, the objective of the module placement stage is to assign to each module a unique location on the chip, so that no two modules overlap and the design constraints imposed by the particular design style (such as row-based placement for standard cells) are satisfied.

The success of the subsequent stages in the physical design process, and the eventual yield, performance and cost of the chip, depend critically on how well the placement is done. The quality of the placement can be measured in terms of the effect it has on the routing procedure. Assuming that the routing algorithm itself is good, a high quality placement will lead to a final routing in which the wires are uniformly distributed over the area of the chip. Wire lengths will be short, on the average, thus reducing signal propagation delays and increasing the overall performance of the chip. With short wire lengths, the wiring area of the chip will be minimized, and hence the total area of the chip will be smaller. This leads to increased chip yield and reduced cost. The use of high quality placement algorithms also makes it possible to increase the scale of integration, thus increasing the density and functionality of integrated circuits.

2.1. Previous Approaches to the Placement Problem

Although it is easy to define the quality of a placement in terms of the results obtained *after* routing the interconnections, there is no easy way to formulate these objectives mathematically in such a way that a placement can be evaluated *before* the routing is performed. The placement is usually done using some modified objectives that approximate the true objectives outlined. However, since these objectives are only approximate, the placement and routing procedures may need to be iter-

ated several times to obtain a good design. This underscores the need for improved placement objectives that accurately reflect the final objectives, and the need for faster solution techniques for the placement and routing problems.

The most common objective used during the placement process is to minimize the total interconnect length, since shorter wire lengths can be expected to result in smaller and faster chips, on the average. The following sections describe some of the commonly used approaches for placement.

2.1.1. Constructive and iterative approaches

Several algorithms for module placement are described in Ref. 1. The algorithms are classified into two broad categories: constructive approaches and iterative improvement approaches. Constructive approaches build up a placement in a manner similar to the way a human designer would. However, the "rules" used intuitively by a human designer are extremely difficult to incorporate into a computer program, and such approaches usually produce poor results. Iterative approaches, on the other hand, start with an initial placement and attempt to improve it by iteratively perturbing the current placement until no further improvement can be found. These approaches usually produce better results.

2.1.2. Stochastic approaches

In the 1980s, a new class of general-purpose *stochastic* optimization algorithms was developed, which proved to be very effective in finding good solutions to computationally complex combinatorial problems. These approaches are similar to iterative approaches in that they begin with an initial solution, which is improved as the algorithm proceeds. The difference lies in their attempt to avoid locally optimal solutions (which cannot be further improved by any perturbations or *moves*) by randomly allowing moves that increase the cost of the solution. These moves give the algorithm a chance to reach the globally optimum solution.

The best known of this class of algorithms is "simulated annealing".[2] TimberWolf,[3] an annealing-based placement and global routing package, is an example of a highly successful application of annealing to physical VLSI design. One of the main drawbacks of simulated annealing is the large amount of computational effort required to obtain good solutions. In Ref. 4, another stochastic optimization technique called Simulated Evolution was used for the placement of standard cells. A related technique called Stochastic Evolution[5] was also applied to the standard cell placement problem and found to produce results competitive with annealing, with less computational effort.

2.2. Min-Cut Placement

In most of the placement approaches described in the previous section, the objective was to minimize the total (or average) wire length. As mentioned in Ref. 1, it is observed that placements generated by such algorithms often lead to non-

uniform routings, with crowded routing channels particularly near the center of the chip. Thus, the original objective of minimizing the chip area will not be satisfied, even though the wires are short. This observation led to the development of a new class of placement approaches, called "min-cut" placement.[6] In these congestion-sensitive approaches, the objective is to minimize the number of nets crossing a set of horizontal and vertical cut lines crossing the chip. The min-cut partitioning approaches lead to placements with short wire lengths by finding strongly connected groups of modules, so that in the final placement, most of the wires do not travel long distances. In fact, minimizing the total number of nets cut by a regular rectangular grid of cut-lines is equivalent to minimizing the total wire length of all the nets, based on the semiperimeter wire length model.[6] However, most min-cut approaches do not minimize the *total* number of nets cut by all cut lines simultaneously; instead, the cut lines are introduced sequentially, and the minimization of the number of nets cut by each of them is constrained by the partitioning induced by the earlier cut lines. As a result, if the initial cut lines are at the center of the chip, the connections are "pushed away" from this region, so that the eventual routing is more uniform than standard wire-length minimization techniques.

Several sequential min-cut placement algorithms have been developed,[6,7,8] most of these differing mainly in the position and sequence of the cut-lines. Two popular approaches are *bisection* and *quadrisection*. In Ref. 7, a min-cut bisection approach to standard cell placement is described, in which the chip is first bisected by a horizontal cut-line, and the resulting rectangles are bisected by vertical cut-lines. The procedure is then applied recursively to the four resulting squares, until the squares are so small that they contain only a few cells each.

In Ref. 8, the bisection approach is extended to be better suited for two-dimensional standard cell placement problems. Partitioning is performed along two perpendicular cut lines simultaneously, so that the chip is divided into four blocks in a single step. Each block is then recursively quadrisected, until each cell has been assigned to a unique row. The exact location of cells within the rows is then found by bisecting the blocks by vertical cut-lines until each block contains at most one cell. The quadrisection approach was found to yield better results than bisection, because of its two-dimensional nature. The area of the chip after global routing was found to be competitive even with simulated annealing.

Sequential min-cut placement algorithms suffer from a common problem of lack of "global information." The partitioning of a block at a particular level of the hierarchy must not be done independently of the other blocks, since the optimal partitioning of a block depends on the cells in the other blocks, and the way in which they are partitioned. This problem was partially overcome by a technique called "terminal propagation",[7] in which connections to cells outside the block currently being partitioned are "propagated" to the boundary of the block, thereby influencing the partitioning (Fig. 1). However, since the blocks at a given level of the hierarchy are partitioned in a sequence, the blocks that are partitioned earlier have less detailed external information than those that are partitioned later. Thus,

it may be necessary to iterate the partitioning of all the blocks at each level of the hierarchy a number of times, until a stable solution is reached.

In the next section, we show how the placement problem can be formulated as a hierarchical binary decision problem and solved using an analog network that belongs to a family of densely interconnected networks known as Artificial Neural Networks. It will be shown that the binary formulation is essentially equivalent to the min-cut quadrisection approach. In addition to the speed advantage of using a neural network, for the particular case of module placement, the parallelism of the network makes it possible to efficiently incorporate a "global view" into the network, eliminating the need for techniques such as terminal propagation.

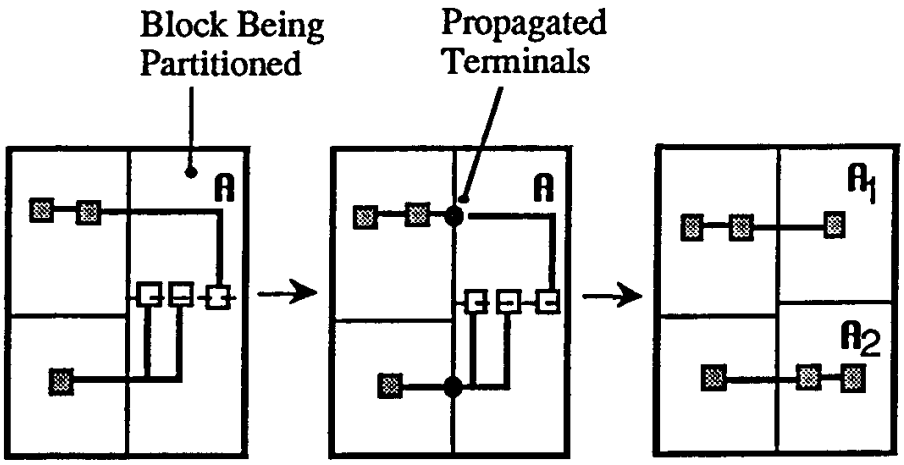


Fig. 1. Terminal propagation.

2.3. Artificial Neural Networks and Optimization

In recent years, there has been a resurgence of interest in so-called "Artificial Neural Networks" (ANNs),[9] which are essentially networks of nonlinear analog devices called "neurons," modeled on the networks of biological neurons in animal and human brains. The advantage of using a neural network as a computational device is that computations are performed in parallel by a large number of simple analog processors. The massive parallelism and analog-domain computations can drastically reduce the time taken to find a solution, in comparison to sequential digital computers. The analog networks can converge to solutions in a few time-constants, which may be typically microseconds to milliseconds. This can potentially make them an effective alternative to standard sequential computers in performing such computationally intensive tasks as pattern recognition and the solution of optimization problems. Advances in VLSI technology have made hardware implementations of these networks feasible, bringing the networks closer to the domain of practical applications. In addition, the distributed nature of the storage and processing of information in these networks gives them a degree of robustness that cannot be found

in conventional computers, making them appealing for situations where reliability is a concern.

2.3.1. The Hopfield network

In 1982, a new neural network architecture was introduced, with interesting collective computational properties that were a direct consequence of its architecture. In its original form,[10] the network consists of a number of "neurons" which communicate with each other via a matrix of interconnections (Fig. 2). The neurons are two-state elements with a step or signum relationship between input and output. Each neuron also has an independent current source at its input, which effectively controls the threshold of the neuron.

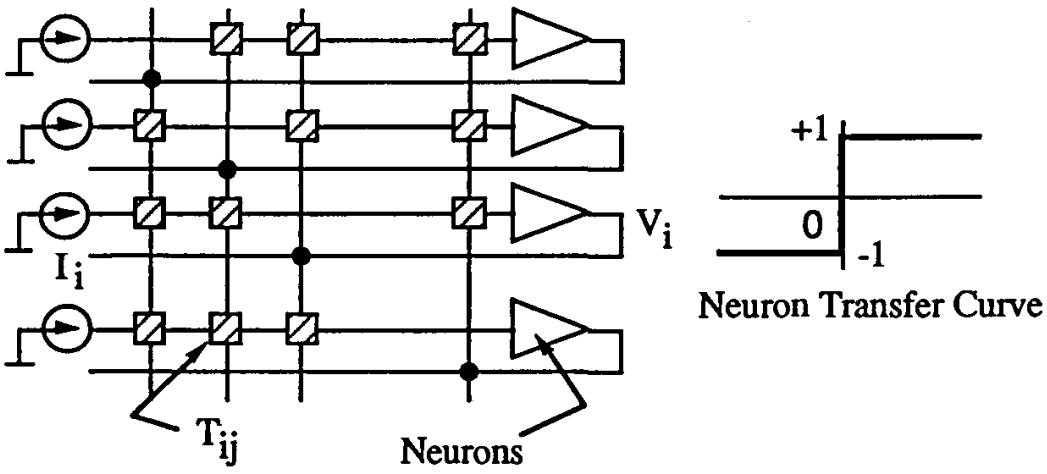


Fig. 2. The Hopfield network.

The input U_i to a neuron i is a linear combination of the output voltages V_j of all the neurons, plus the current source I_i,

$$U_i = \sum_{j=1}^{n} T_{ij} V_j + I_i \tag{1}$$

where T_{ij} is the weight of the connection from neuron j to neuron i.

Consider the following quadratic functional of the state vector V:

$$E = -\frac{1}{2} \sum_i \sum_j T_{ij} V_j V_i - \sum_i I_i V_i \ . \tag{2}$$

The quantity E is frequently called the "Energy" functional in neural network literature, and it can be used to show that the network is stable, under the following conditions:

1) The interconnection matrix **T** is symmetric and has zeroes on the diagonal.
2) The neurons update their states asynchronously, i.e., at independent instants of time.

If these conditions are satisfied, the network can be shown to be stable, and any stable state of the network constitutes a local minimum of E.

In a later paper,[11] it was shown that a physically realizable version of this network preserved the important properties of the idealized network. The new version of the network has neurons with a smooth sigmoidal input-output relationship instead of a step function (Fig. 3). The interconnection strengths are represented by conductances (negative interconnection weights are handled by inverting neuron outputs), and time delays in the physical devices used to build the neurons are taken into account by capacitances at the inputs of the neurons. As the transfer curve of the neurons becomes steeper, the minima of the energy function of the analog network move toward the minima of E. In the high gain limit, the only stable states of this continuous-time, continuous-state system coincide with the stable states of the simpler discrete-time, discrete-state system. Therefore, in the following, our attention will be restricted to the discrete model, since it is easier to simulate on a computer.

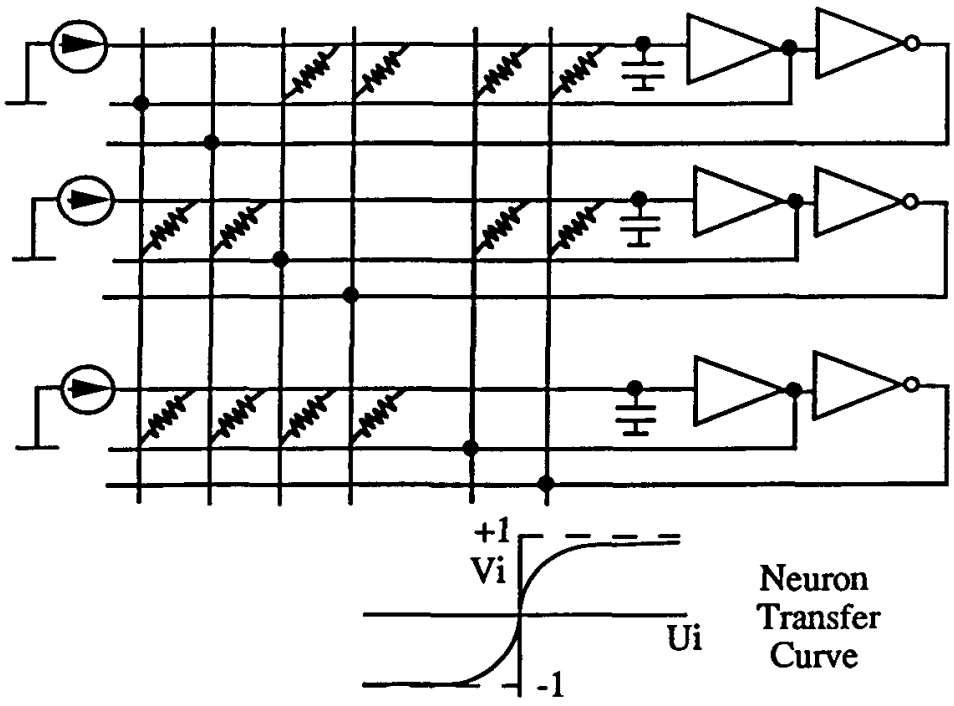


Fig. 3. A physically realizable Hopfield network.

2.3.2. Optimization using a Hopfield network

The first step in solving an optimization problem using the Hopfield network is to find a suitable representation of the problem in terms of the state of the network.

By this, we mean that a correspondence or mapping must be found between the set of states $\{V^i\}$ of the neural network and the set of all solutions $\{S^i\}$ of the problem. The mapping does not necessarily have to be one-to-one, but every solution S^i must be represented by at least one state.[12,13]

Let the cost to be minimized (or maximized) in the problem be $C(S^i)$, and let V^i be one of the network states corresponding to the solution S^i, under the given mapping. Then,

1) There must exist a matrix T and a vector I such that $C(S^i) = E(V^i) + a$ constant, where E is the quadratic energy function.
2) If there are two states, V^1 and V^2, corresponding to the same solution, then $E(V^1) = E(V^2)$.

If these conditions are satisfied, the next step is to construct a Hopfield network with the appropriate connection matrix and current vector, initialize the network to a random state, and let the network converge to a stable state. This stable state, if it maps to a valid solution, will correspond to a solution of locally optimal cost. (Techniques to avoid local minima will be described later.)

Usually, it is not possible to find a one-to-one correspondence between the states and the set of all valid solutions to the problem. In such cases, the set of states is larger than the set of all solutions, and many states will correspond to "invalid" solutions, i.e., solutions that do not satisfy the constraints of the original problem. In such situations, a "penalty function" $P(V)$ has to be added to the cost of a state, which satisfies the following conditions:

1) $P(V^i)$ is zero for all states V^i that correspond to valid solutions, and positive for those states that do not correspond to valid solutions.
2) $P(V^i)$ is a quadratic function of V^i, so that the total energy function, $E + P$, is still a quadratic function.

The addition of this penalty function changes the values of the interconnection matrix and current vector. The new network tends to avoid invalid states, and converges to valid solutions with an increased probability.

2.3.3. Mapping the two-dimensional module placement problem

In order to solve the placement problem using a Hopfield network, we need to find a mapping such that placements are represented as binary vectors, and the objective function is of the same form as E. Since the size of the solution space of the placement problem is $O(n!)$, where n is the number of modules, a lower bound on the number of two-state neurons required to completely represent the problem is $O(n \log n)$.

One approach models the placement problem as an *assignment* problem, in which n modules are assigned to unique locations on a regular grid of n points.[14] The approach uses n^2 neurons, indexed as $N_{ij}, i, j = 1, \ldots, n$. The output of neuron N_{ij} is 1 if module i is assigned to location j, and 0 otherwise. This mapping

permits the formulation of an appropriate objective function based on minimizing total wire length. However, the number of neurons needed is n^2, and the number of programmable interconnections between the neurons is $O(n^4)$. Clearly, it would be virtually impossible to construct a neural network in hardware to solve even a moderately large practical problem using this formulation.

In the *hierarchical binary approach* presented here, a neural network with $O(n)$ neurons is used in $O(\log n)$ stages to arrive at the final solution. The x and y positions of each module are computed one bit at a time, starting with the most significant bit. In the first stage, the neural network solves the binary decision problem of deciding which quadrant of the chip each module should be placed in. In the next stage, the same problem is solved *within each quadrant* of the first stage. The process continues recursively until each module has been assigned to a unique location. In each stage, the decisions are made with the objective of keeping strongly connected modules in the same quadrant, and with the constraint that the quadrants are of approximately equal area.

This approach is essentially equivalent to solving the min-cut quadrisection problem using a neural network. In the details of the neural network mapping that follow, we will therefore use the terminology of graph quadrisection and min-cut placement.

The aim of graph quadrisection is to divide the set of vertices of a graph G into four equal sets of vertices, such that the number of edges connecting vertices in different sets is minimized (Fig. 4). A significant difference between quadrisection and 4-way partitioning of a graph is that quadrisection is a two-dimensional problem: the four sets of vertices can be imagined to lie in a plane, at the four corners of a rectangle. Edges connecting vertices lying in diagonally opposite sets contribute more to the cost of the partition than do edges that connect vertices in sets lying on the same edge of the rectangle. The total cost of a partition thus has two components, the *vertical cost* and the *horizontal cost*. In Fig. 4, edge uv contributes to the horizontal cost, vw contributes to the vertical cost, and uw contributes to both the horizontal and the vertical costs. To formulate the objective function mathematically, we associate two *binary* variables p_{ix} and p_{iy} with each vertex i in the graph. The values assumed by these variables are $+1$ or -1, depending on the set to which i belongs.

It is possible to weight the horizontal and vertical costs differently. This may be desirable when the aspect ratio of the rectangle is not unity, or if, for some reason, it is preferable to have more edges cut in one direction than in the other (perpendicular) direction. If we define A to be the ratio of the vertical to the horizontal cost, we can write the total cost as

$$\text{Total Cost} = -\frac{1}{2}\sum_i \sum_{j \neq i} C_{ij} p_{ix} p_{jx} - \frac{1}{2}\sum_i \sum_{j \neq i} (AC_{ij}) p_{iy} p_{jy} \ . \qquad (3)$$

In the standard version of the quadrisection problem, the minimization of this cost has to be done subject to the constraint that the number of vertices in each

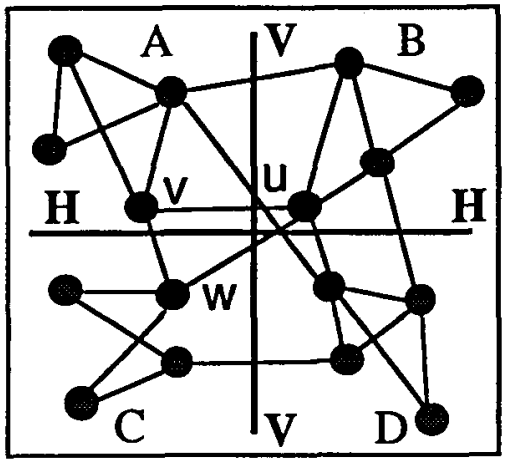


Fig. 4. The graph quadrisection problem.

set is equal. However, when applied to min-cut placement, the constraint has to be modified to account for the fact that the modules may vary in size, and it is not possible, in general, to partition the set of modules such that the sizes of the resulting sets are identical. Therefore, we must seek to minimize the deviation in the sizes of the sets instead. With each vertex i, we associate a value α_i which represents the size of the vertex. Then minimizing the deviation in the sizes of the sets is equivalent to simultaneously minimizing all three of the following quantities:

$$X\text{-deviation} = \left(\sum_i \alpha_i p_{ix}\right)^2 = \sum_i \sum_{j \neq i} \alpha_i \alpha_j p_{ix} p_{jx} \ , \tag{4}$$

$$Y\text{-deviation} = \left(\sum_i \alpha_i p_{iy}\right)^2 = \sum_i \sum_{j \neq i} \alpha_i \alpha_j p_{iy} p_{jy} \ , \tag{5}$$

$$Q\text{-deviation} = \left(\sum_i \alpha_i p_{ix} p_{iy}\right)^2 = \sum_i \sum_{j \neq i} \alpha_i \alpha_j p_{ix} p_{jx} p_{iy} p_{jy} \ . \tag{6}$$

These expressions can be added to the total cost function as penalty functions, after weighting them by appropriate penalty multipliers λ_x, λ_y, and γ. Incorporating all these terms, the quantity to be minimized can be written as

$$\begin{aligned} \text{Cost} + \text{Penalty} = & -\frac{1}{2} \sum_i \sum_{j \neq i} C_{ij} p_{ix} p_{jx} - \frac{1}{2} \sum_i \sum_{j \neq i} (AC_{ij}) p_{iy} p_{jy} \\ & + \lambda_x \sum_i \sum_{j \neq i} \alpha_i \alpha_j p_{ix} p_{jx} + \lambda_y \sum_i \sum_{j \neq i} \alpha_i \alpha_j p_{iy} p_{jy} \\ & + \gamma \sum_i \sum_{j \neq i} \alpha_i \alpha_j p_{ix} p_{jx} p_{iy} p_{jy} \ . \end{aligned} \tag{7}$$

2.3.4. The modified Hopfield network

The expression of (7) cannot be minimized directly by a Hopfield network, since it contains quartic terms in the penalty function. We now seek a network architecture, based on the Hopfield network, that can minimize this function. Some insight into the structure of such a network can be gained by rearranging the terms of (7). The first and third terms, and the second and fourth terms, are grouped together. The last term is split into two and added to the first and second terms, yielding

$$\text{Cost} + \text{Penalty} = -\frac{1}{2}\sum_i\sum_{j\neq i}(C_{ij} - \lambda_x\alpha_i\alpha_j - \gamma_x\alpha_i\alpha_j p_{iy}p_{jy})p_{ix}p_{jx}$$

$$-\frac{1}{2}\sum_i\sum_{j\neq i}(AC_{ij} - \lambda_y\alpha_i\alpha_j - \gamma_y\alpha_i\alpha_y p_{ix}p_{jx})p_{iy}p_{jy}\ . \qquad (8)$$

This expression can be interpreted as the total energy function of *two coupled Hopfield networks*. The two networks, termed the *X*- and *Y-subnetworks*, each solve one "dimension" of the two-dimensional placement problem, and the coupling between them ensures that the solutions to the one-dimensional problems satisfy the two-dimensional constraints of equal quadrisection. The two subnetworks are coupled in the following way: the matrix formed by the *outer product* of the state vector of the Y-subnetwork is weighted by the factor γ and subtracted from the interconnection matrix of the X-subnetwork. The interconnection matrix of the Y-subnetwork depends on the state of the X-subnetwork in a similar way. Figure 5 shows a symbolic representation of the modified network.

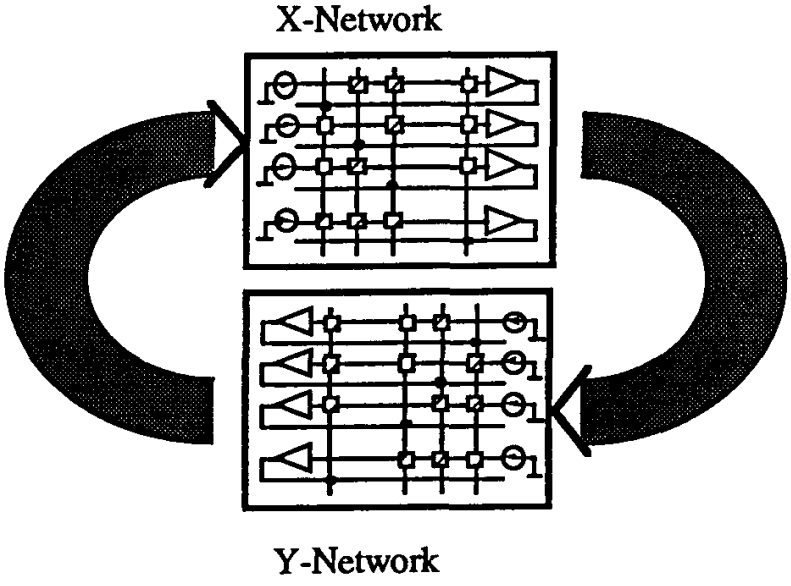


Fig. 5. The modified Hopfield network.

The proof of stability of the modified Hopfield network is similar to that of the original network.[10] Consider, for simplicity, the energy function of (8), with all the

α_i's equal to one:

$$E = -\frac{1}{2}\sum_i \sum_{j \neq i} (C_{ij} - \lambda_x - \gamma_x p_{iy} p_{jy}) p_{ix} p_{jx}$$

$$-\frac{1}{2}\sum_i \sum_{j \neq i} (AC_{ij} - \lambda_y - \gamma_y p_{ix} p_{jx}) p_{iy} p_{jy} \tag{9}$$

We assume that the neurons update their state asynchronously, that is, only one neuron can change state at any given instant of time. Suppose that p_{kx} changes to $p_{kx} + \delta p_{kx}$. The change in E is given by

$$\Delta E = -\sum_{j \neq k} (C_{jk} - \lambda_x - \gamma_x p_{ky} p_{jy}) p_{jx} \delta p_{kx}$$

$$+\sum_{j \neq k} \gamma_u p_{ky} p_{jy} p_{jx} \delta p_{kx} \tag{10}$$

which can be written as

$$\Delta E = -\left[\sum_{j \neq k} (C_{jk} - \lambda_x - (\gamma_x + \gamma_y) p_{ky} p_{jy}) p_{jx}\right] \delta p_{kx} \tag{11}$$

If the input to neuron k in the X-network is given by

$$U_{kx} = \sum_{j \neq k} (C_{jk} - \lambda_x - (\gamma_x + \gamma_y) p_{ky} p_{jy}) p_{jx} \tag{12}$$

then ΔE is always nonpositive. The case when a neuron in the Y-network changes state is identical. The result holds even when the α_i's are unequal. Thus, since E is bounded, the network will converge to a stable state that locally minimizes E.

2.3.5. Module placement and addition of global information

The strategy followed for dividing the modules into rows and fixing their position within the rows is similar to that used in Ref. 8. However, there is one significant difference. At the kth stage of the procedure, all 4^k blocks are partitioned *simultaneously*, instead of sequentially. Furthermore, the 4^k quadrisections are not performed independently; they influence each other continuously so as to achieve an overall solution that is optimal in a global sense. A technique similar to terminal propagation can be incorporated very easily into the formulation, and the iterations required during sequential quadrisection are not necessary here, since all blocks at the same level of the hierarchy are processed at the same time. Because of the parallelism of the network, this advantage can be obtained at no additional computational or hardware complexity.

In our approach, the addition of external connection information is achieved in two ways. The first is the use of a novel partitioning scheme, instead of ordinary quadrisection. The second is through the use of independent current sources in the modified Hopfield network.

The difference between quadrisection and our partitioning scheme is illustrated in Fig. 6. In the new partitioning scheme, cut lines 1–4 are processed *simultaneously*, while enforcing the equal partition constraints within the individual blocks. This automatically leads to the addition of external connection information. To see this, consider four cells a, b, c and d, located in blocks A, B, C and D, respectively. Cells b, c and d are connected to cell a by two-terminal nets. Since cells a and b share cut line 1, they will tend to be placed on the same side of the cut line, thus reducing the vertical cost. Similarly, since cells a and c share the cut line 3, they will tend to move to the same side of this cut line, thus reducing the horizontal cost.

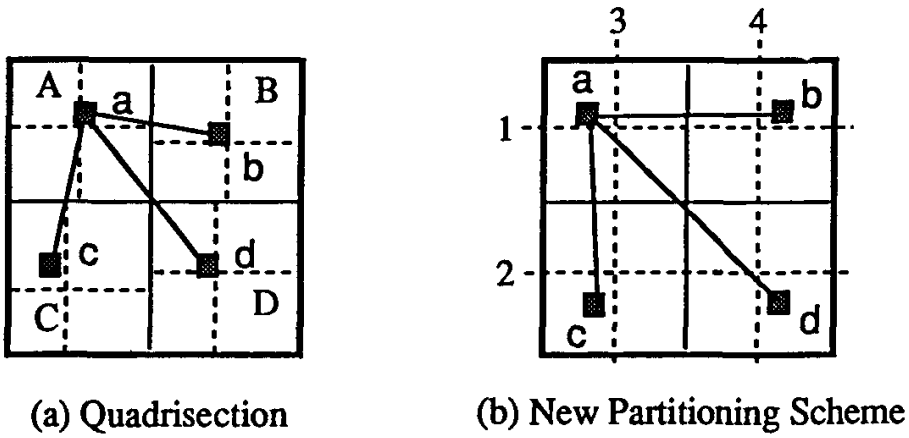


Fig. 6. Difference between quadrisection and the new partitioning scheme.

The external information added by the sharing of cut lines is not complete. For example, although cells a and b are each aware of the vertical position of the other (with respect to cut line 1), there is as yet no information that will cause them to move closer in the horizontal direction. There is a similar problem in the vertical direction with cells a and c. Moreover, since cells a and d do not share any cut lines, there is no connection information between them. To make the external information more complete, we make use of the independent current sources provided at the input of each neuron in the Hopfield network. For example, to bias cells a and b to move closer to each other in the horizontal direction, we add a current source equal to C_{ab} at the input of the neuron corresponding to cell a in the X-network, and a current source equal to $-C_{ab}$ at the input of the corresponding neuron for cell b. For cells a and c, current sources will be added at the inputs of the neurons in the Y-network. For cells a and d, current sources will have to be added to the neurons in both the X- and the Y-networks.

Figure 7 shows the effect of including both sources of external connection information. For this simple example, where there are no conflicting placement requirements, this is obviously the best possible solution. However, in practice, the situation is more complex, and no "best" solution can be easily identified. In such situations, our technique will still exhibit superior performance, since it has more information available to make decisions about trade-offs.

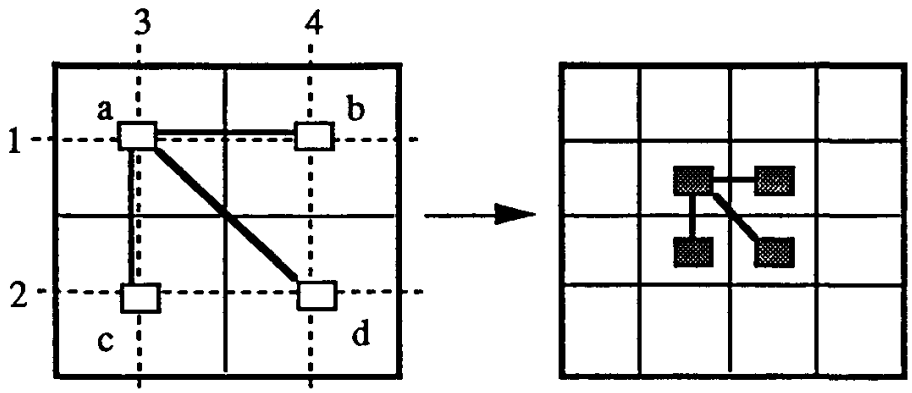


Fig. 7. Effect of adding external connection information.

The simultaneous partitioning of blocks and the natural addition of external connection information are the most significant advantages of using the neural network, besides its parallel computation power. They contribute significantly to the success of our approach.

2.4. Extensions and Results

2.4.1. Multiterminal nets

In typical practical circuits, the interconnections are *multiterminal nets*, in which two or more circuit modules share a common electrical node. This situation cannot be directly represented by a graph model, since in graphs, edges always connect at most two vertices. Multiterminal nets are usually represented using a *hypergraph model*, but this model cannot be handled by a Hopfield network.

One way of representing multiterminal nets using a graph model is to convert each net to a *clique* or fully-connected subgraph, as shown in Fig. 8. Each vertex in the net is connected to every other vertex in the net by an edge, so that the complete connectivity information of the net is maintained. To reduce the bias towards larger nets inherent in the clique model, the edges in the clique are weighted by a factor inversely related to the number of terminals. The average number of edges cut by a partition in the clique model is given by

$$\frac{\sum_{k=0}^{m} \binom{m}{k} k(m-k)}{2^m} \tag{13}$$

which, upon simplification, can be shown to be equal to $m(m-1)/4$. In the hypergraph model, the average cost is equal to

$$\frac{2^m - 2}{2^m} \tag{14}$$

since all except two of the 2^m partitions have a cost of unity, and two have a cost of zero. Therefore, the weight of the connection between two cells i and j in the clique model is given by

$$C_{ij} = \sum_{k:i,j \in N_k} \frac{4(2^{m_k} - 2)}{m_k(m_k - 1)2^{m_k}} \tag{15}$$

where m_k is the number of terminals connected by net N_k.

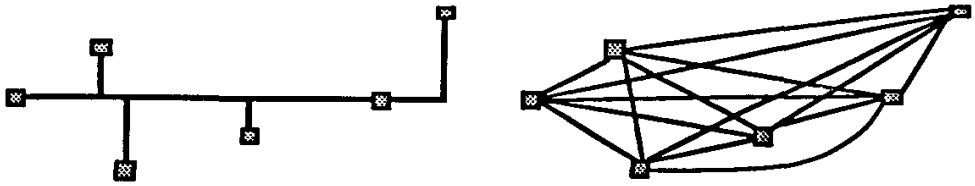


Fig. 8. Clique model for multiterminal nets.

2.4.2. Escape from local minima

The Hopfield network, and its modification described here, can only be guaranteed to converge to local minima of their energy functions. These local minima are often considerably higher in cost than the global minimum. As a result, if the network is being used to solve an optimization problem, the solution may be quite poor. Some of the ways to avoid local minima are described here.

In "Stochastic Neural Networks",[15,16] the technique of "Diffusion," which is the continuous-time continuous-state analog of simulated annealing, is incorporated in the Hopfield network. Uphill moves are allowed by introducing random noise at the inputs of the neurons, so that the dynamics of the network are governed by a set of stochastic differential equations. The amplitude of the noise is reduced slowly, making the network more deterministic, and the network eventually converges to a global minimum under certain conditions, with probability one.

Implementing this network in hardware would involve designing circuits to compute hyperbolic functions, and including on-chip wideband noise generators. A major advantage of this network is the fact that its dynamics are continuous-time and continuous-state, so that a fully parallel analog implementation is possible. Such a network would have the maximum speed advantage over a computer simulation.

Simulating an analog stochastic neural network on a computer is a computation-intensive task. However, the simulated annealing algorithm can be incorporated into a Hopfield network simulator in a very straightforward manner. The state space is the power set $2^{\mathbf{V}}$ of the binary vector $\mathbf{V}$ of neuron output voltages. Moves are generated by flipping the output V_k of a randomly chosen neuron k. The change in energy ΔE due to a move is simply given by

$$\Delta E = -U_k \Delta V_k \tag{16}$$

Since the U_k's (which are the inputs to the neurons) are already available in the simulator, the change in energy caused by any move can be computed easily. This discrete-time discrete-state modeling of the analog stochastic network is sometimes called a Boltzmann machine,[17] and its global-minimum seeking properties are similar to those of a stochastic neural network.

Another technique for avoiding local minima, The Mean Field Theory method, is based on a technique in statistical physics, in which a set of deterministic equations is used to approximate the statistical behavior of a complicated system. This technique, being deterministic, is computationally less expensive than simulated annealing, but is approximately equivalent to it in theory, and yields comparable results in practice.[12,18]

The diffusion machine and the mean field theory technique are both approximately equivalent to simulated annealing in theory. The results quoted in the next section were obtained by computer simulations of a Boltzmann machine, and should be reproducible by hardware implementations that use one of the other two techniques.

2.4.3. Adaptive penalty multiplier control

It was found that the values of the penalty multipliers had a significant effect on the quality of the final placement. To ensure high quality placements, an adaptive feedback control scheme was developed to continuously adjust the multipliers and keep them at their optimal values. A similar technique has been independently developed for the TimberWolf 5.4 placement package.[19] The algorithm essentially consists of a negative feedback loop, in which the outputs (the X-,Y- and Q-deviations) are continuously monitored, and the error (the difference between the outputs and the target values) is fed back and used to update the inputs (λ_x, λ_y and γ). The feedback loop works in conjunction with the annealing process; at high temperatures, the average energy is high, and therefore, the average penalty can also be expected to be high. As the system cools, the penalties should move closer to their target values. To account for this kind of behavior, the target penalties are made temperature dependent. After each temperature step, the values of the multipliers are updated depending on whether the actual penalties are greater than or lesser than the target penalties.

2.4.4. Placement results on industrial standard cell circuits

The performance of the neural network placement approach was tested on a set of eight standard cell circuits, including a 752-cell benchmark circuit distributed at the ACM/IEEE Physical Design Workshop in 1987. The placement results were compared with results obtained from the simulated annealing-based placement package, TimberWolf 5.4 (Ref. 20). The comparisons are shown in Table 1. It must be emphasized that these comparisons are not included for the purpose of demonstrating the superiority of either method: they were performed to investigate the feasibility of the neural network approach. A comparison of the execution times for the two programs is not relevant here, since our program is a functional simulation of the neural network. The neural network implemented in hardware will be much faster.

The placements have been compared using four different metrics, three of which are wire length estimates before global routing, and the last is a wire area estimate obtained after performing the global routing, using the TimberWolf global router. (The router did not work on some of the examples, which is why some of the entries in the last column are missing.) The semiperimeter estimate is the objective minimized by TimberWolf. The rectilinear clique estimate corresponds approximately to the objective minimized by the neural network. The straight trunk estimate has also been included, as it is a good model for multiterminal nets.

From Table 1, it is apparent that when the average number of terminals per net is large, there is a wide variation between the different net length metrics. The semiperimeter length is usually an underestimation of the actual length of a net with many terminals; the straight trunk model is a better estimate in such cases.

Table 1. Placement results using different metrics.

Circuit	Number of Cells	Avg. pins/net	Semi Perimeter	Rect. Clique	Straight Trunk	Routing Area
try	60 (4 rows)	7.37	+2.7%	+1.72%	+0.25%	+4.87%
try	60 (6 rows)	7.37	+13.1%	+7.5%	+0.19%	+2.85%
ex1	183 (5 rows)	3.91	+6.6%	+1.36%	−0.33%	+4.55%
t200g	200 (8 rows)	9.47	−2.7%	−5.7%	−2.3%	—
cir	286 (7 rows)	3.83	+0.54%	+5.32%	+3.79%	+6.95%
ckta8	469 (8 rows)	3.20	+10.2%	+1.76%	+6.1%	—
p1	752 (17 rows)	3.29	+15.2%	+9.76%	+10.9%	+5.87%
ckt5	800 (8 rows)	4.06	+15.2%	+11.6%	+3.47%	+4.23%
t1000g	1000 (16 rows)	14.5	+21.5%	+10.2%	+9.5%	—

(All percentages are w.r.t. TimberWolf 5.4 results. A positive percentage indicates the degree of additional wire length or area over the TimberWolf value.)

The straight trunk wire length using our approach is very close to that obtained using TimberWolf, typically within 10%. For circuits having a high average number of terminals per net, such as the 1000 cell circuit and the 6-row placement of circuit *try*, the comparison figures are considerably better when the more realistic straight trunk metric is used.

In terms of the estimated routing area after global routing, which is closest to the true final objective of minimizing chip area, the performance of the neural network is very close to that of TimberWolf. Also, the performance is uniform and does not worsen with increasing problem size. Considering the limitations imposed by the particular architecture, the Hopfield network approach renders excellent results. The results can be expected to be even better if the neural network is implemented in hardware. The parallel min-cut algorithm will be truly executed in parallel, instead of the pseudo-parallel execution in a computer simulation. The analog neurons will be more capable of avoiding local minima, and longer annealing schedules can be used. The feedback control scheme will also be more effective since it can operate continuously. Empirical parameter values can be found by running several trials, since each trial will take much less time. Also, since the rectilinear clique wire length estimate minimized by the network has a higher correlation with the final routing area, the network can be expected to find placements with smaller routing areas.

2.5. Discussion

2.5.1. Limitations of the neural network approach

The advantages of using a neural network have been discussed in earlier sections of this paper. However, using a neural network as an optimization tool has certain limitations as well. One major limitation is that the network must be implemented in hardware to realize its full potential. The implementation of a Hopfield network involves a trade-off between computational complexity and hardware complexity. However, a one-time investment in hardware results in recurring savings in computation time.

In our application to cell placement, the restrictions imposed by the structure of the neural network force us to represent multiterminal nets using a clique model. This may not be the best model to use; for example, the straight trunk model may be more realistic, but it cannot be mapped onto the network. Restrictions such as these may lead to suboptimal mappings of problems onto the network.

It is difficult to incorporate heuristics, based on human experience with the problem to be solved, into the neural network. As a result, the search performed by the neural network in the solution space tends to be random and undirected, as opposed to that performed by a typical algorithm on a conventional computer, where the search can be guided by rules and heuristics. However, the advantage of parallel analog computation can offset this drawback by allowing more exhaustive searches.

2.5.2. Hardware implementation

The success of the placement approach described in this section hinges on the availability of hardware implementations of large neural networks. In this sense, the approach is futuristic, since it is not yet possible to implement truly large networks (with several thousands of neurons) in VLSI.

The main problem in hardware implementation of neural networks is the large number of programmable analog interconnections or *synapses* required. In a Hopfield network, the number of interconnections grows as the square of the number of neurons. Many different approaches for implementing these interconnections have been suggested, such as fully analog,[21,22,23] hybrid digital-analog[24,25] and switched capacitor.[26] Other possibilities are fully digital implementations, using numerical integration techniques, or pulse-width modulation techniques. However, it does not seem likely that any of these techniques will permit networks with over a thousand neurons to be integrated onto a single chip. For large networks, multichip modules (MCMs) with several smaller network chips interconnected via specialized interconnect chips, all mounted on the same ceramic substrate, could be useful. New technologies, such as ULSI or opto-electronic integrated circuits, may also provide good solutions.

The modification to the Hopfield network architecture poses more challenges for hardware implementation. The outer-product matrix generated by the output vector of one subnetwork has to be added to the connection matrix of the other subnetwork. This architecture suggests a multilayer implementation, in which the two subnetworks occupy two separate layers, with two additional layers between them for the outer-product matrices. Thus, the overall package could be a multichip module, with two chip layers.

The global optimization schemes, as well as the penalty multiplier control loop, also need to be implemented in hardware. The stochastic network approach requires the implementation of on-chip independent noise-sources, while the Mean Field Theory approach requires analog neurons with continuously adjustable gain. The feedback control loop requires that the synapse weights be continuously variable, which implies that analog synapses would be preferable to digital ones.

2.5.3. Application to other VLSI CAD problems

An investment in neural network hardware would be more worthwhile if the network could be used as a general-purpose hardware "engine" to solve a variety of problems. If this were not the case, it would be a better idea to invest in developing a hardware realization of a general-purpose optimization tool such as simulated annealing.

A number of recent papers have presented promising results on different VLSI combinatorial problems using the Hopfield neural network model and modifications of it. Graph and hypergraph partitioning problems occur frequently in all stages of VLSI design. The applicability of the Hopfield network to graph partitioning

has been described in this section. A modification to the Hopfield network that can handle circuit (hypergraph) partitioning is described in Ref. 27. In Ref. 28, the module orientation and rotation problems have been solved using Hopfield networks. In Ref. 29, the global routing problem has been mapped onto the Hopfield network. In the light of these results, and of the results presented in this section, it is apparent that hardware implementations of neural networks can be useful as multipurpose optimization tools for VLSI CAD.

3. Sea-of-Gates Global Routing

In recent years, the Application Specific Integrated Circuit (ASIC) market has expanded rapidly, due to the introduction of Sea-of-Gates (SOG) arrays.[30] Gate arrays and SOG arrays account for roughly half of the ASIC market, and provide the advantages of quick turnaround times, high packing density and high performance.

The SOG array structure introduced by Duchene and Declercq[30] is illustrated in Fig. 9. The SOG rows are separated by the power lines, V_{dd} and ground, and each

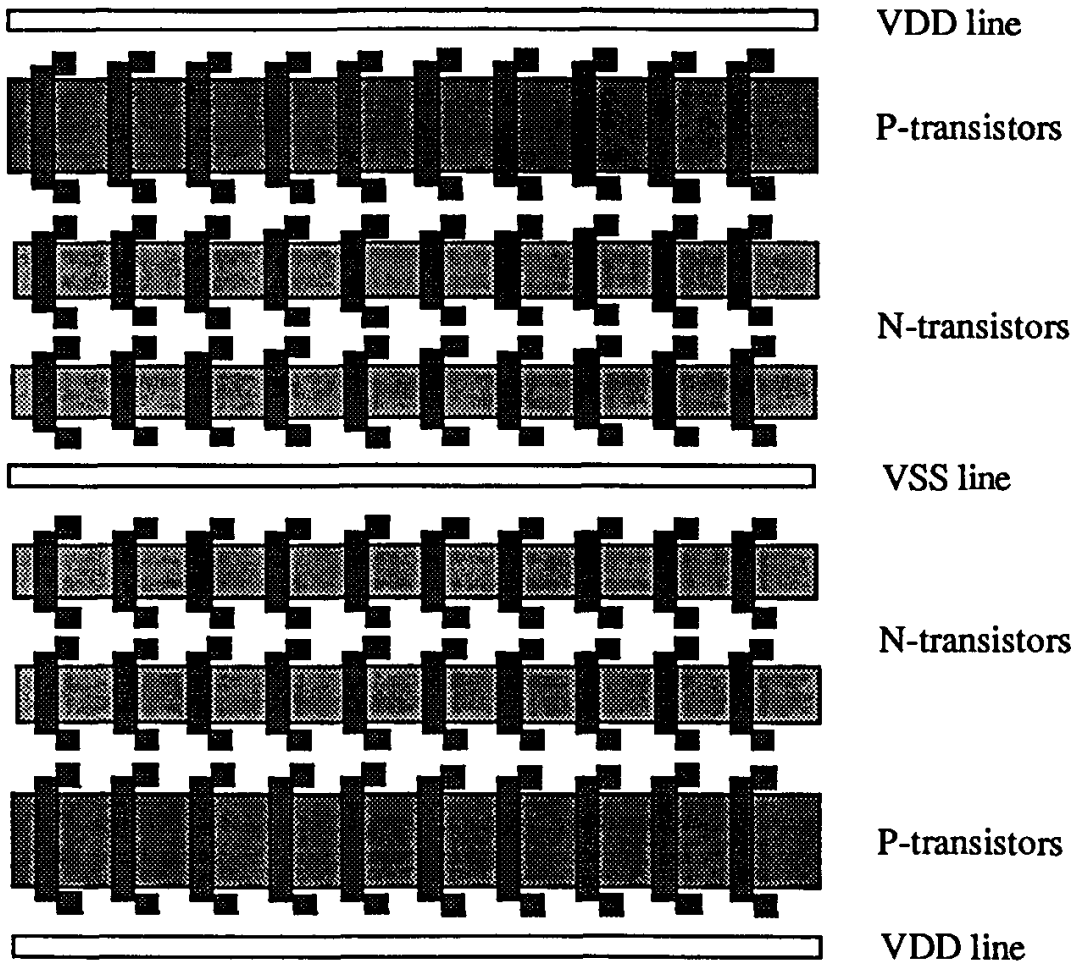


Fig. 9. Sea-of-gates design style.

SOG row has one row of p^+-diffusion and two rows of n^+-diffusion. Intramodule connections are made using vertical (bottom level) metal-1 lines and horizontal (top-level) metal-2 lines. This structure has many of the features that are typically found in most SOG array structures: high packing density, gate-isolation and a truly channelless structure. Moreover, it can support a number of different types of circuits, such as random and regular logic, and dynamic CMOS and analog circuits. For this reason, the global routing scheme described here[31] has been devised based on this structure. It can be extended to other gate array structures as well, if the array and intracell information is extracted properly.

3.1. Motivation for the new routing scheme

Global or "loose" routing is the step that follows module placement in the VLSI physical design process. In this step, the area of the chip is divided into a number of *tiles* or *global routing cells* (GRCs). Terminals of nets are usually assumed to be located at the center of their respective GRCs, so that the area of the chip can be represented as a grid graph. Each edge in the grid graph is assigned a *capacity* which indicates the number of routing tracks that can pass through the corresponding GRC boundary. The aim of the global routing problem is to find, for each net, a tree on the grid graph that connects all the terminals of the net, such that the constraints imposed by the edge capacities are satisfied. There may be a number of other constraints, such as net length or timing constraints, as well.

The global routing problem occurs in most design styles, although the details of the problem may vary. SOG array structures are typically large and channelless, making it difficult for conventional routing schemes to handle their complexity. The goal of the work described here was to devise a router that could take into account the unique features of SOG arrays, and be able to handle very large size arrays efficiently. The development of the router was motivated by the following factors:

(i) *Net-ordering dilemmas.* In VLSI routing, a common reason for poor performance of many routing algorithms is the fact that they process the nets in a sequence, and are thus dependent on the ordering of the nets. A badly chosen ordering can lead to a poor solution. Finding a good ordering of nets is a difficult problem. However, the use of a zero-one Integer Linear Programming (ILP) formulation gets around this problem by considering all nets *simultaneously.*

(ii) *Difficulty in constructing Rectilinear Steiner Trees (RSTs).* The problem of constructing RSTs with five or more terminal pins is NP-complete. Although a good linear time heuristic for constructing RSTs is known,[32] the RST found is constrained to lie within the rectangle bounded by the terminals of the net. This unnecessary constraint may make it very difficult for the router to find an obstacle-free path in channelless SOG arrays. The new router avoids the RST construction step altogether, by constructing each net a few segments at a time, until all the terminals have been connected. This is achieved by

decomposing the global routing process into different phases to handle routing of different kinds of path segments — vertical, horizontal and unrestricted. Even though the resulting spanning tree may not be the shortest possible, it is optimal in a global sense, since it has been constructed by taking all other nets into consideration, via the ILP formulation.

(iii) *Implementation of a Meet-in-the-Middle approach.* A "Meet-in-the-Middle" approach combines the best features of conventional top-down and bottom-up approaches into a hybrid approach. In the new router, the hybrid retains the "divide-and-conquer" advantage of a top down approach, while avoiding the problem of the lack of a detailed picture of the routing at high levels of the hierarchy.

(iv) *Via minimization.* Vias tend to reduce the yield of chips, since each via corresponds to a contact cut in the fabrication process. Since a via is introduced each time a wire bends, nets should be routed with straight lines whenever possible. The new scheme tries to reduce the number of vias by attempting to connect nets using straight line paths initially.

(v) *Performance-driven layout.* To obtain the maximum advantage from high-performance SOG structures, the routing should be done with consideration given to timing constraints on the nets. This feature is incorporated into the new router by associating a priority value with each net, so that timing-critical nets are given preference and routed in the best possible way.

3.2. Zero-One Integer Linear Programming

The zero-one ILP problem is a special case of a linear programming problem in which all the variables in the problem are constrained to take on a binary value of either 0 or 1. 0-1 ILPs have been applied to a number of optimization problems, including the Knapsack Problem, the Traveling Salesperson Problem, and the Assignment Problem.[33]

The mathematical formulation of a 0-1 ILP is as follows:

MAXIMIZE

$$\sum_{j=1}^{n} c_j x_j$$

SUBJECT TO

$$\sum_{j=1}^{n} a_j x_j \leq k$$

AND

$$x_j = 0 \text{ or } 1, \quad j = 1, 2, \ldots, n \ .$$

Algorithms such as the pivot and complement heuristic[34] and branch-and-bound methods[35] have been used successfully for solving 0-1 ILPs, and are used in the Zero-One Optimization Methods (ZOOM) software based on the XMP linear programming library.[36]

In Ref. 37, the 0-1 ILP was suggested as a tool for solving the circuit routing problem. The basic idea is as follows: a number of possible routes are identified for each net. Each route is assigned a *benefit value* which reflects the relative advantage of using that route over the others. All the routes are represented in the ILP as binary variables, and the objective of the ILP is to select one route for each net so as to maximize the total benefit value over all the nets, while satisfying the constraints of routing track availability. All the routes assigned a value of 1 in the ILP solution are finally used.

There have been a few attempts in the past to use ILPs for global routing,[37,38,39] but they have left some details unanswered. There are problems that arise from net decomposition, track sharing and the occurrence of cycles in the routing. There is also the problem of complexity: VLSI global routing problems may lead to ILPs that are too large to be solved directly.

The new SOG global routing scheme, which uses 0-1 ILPs for optimization in all the routing phases, is formulated to take into account the unique features of the SOG environment, and also to avoid the problems encountered by other ILP routers. The ILP constraints are imposed by over-the-cell routing track availability, via availability, and general circuit routing constraints such as the avoidance of loops and the selection of at most one path for a net. The complexity problem is handled by hierarchically decomposing the routing area into manageable slices.

3.3. The Global Routing Phases

The SOG router is designed to use three layers of metallic interconnects. The algorithm is divided into an information extraction step, followed by three distinct global routing phases, as shown in Fig. 10. All three routing phases interact with the 0-1 ILP solver ZOOM (although the approach allows any other ILP package to be used).

During the information extraction step, a two-dimensional array of global routing cells (GRCs) is formed from the netlist, placement results with terminal locations, information pertaining to the SOG array structure, and information about preroutings or blockages in the macrocells. The vertical pitch of the GRCs is determined by the number of horizontal tracks within each row of the SOG base array, so as to preserve the natural row structure of the array. The horizontal pitch of the GRCs is user-defined. Since each n-p diffusion pair for CMOS circuits is explicitly represented by two subrows within a row of GRCs, the number of rows in the array is determined from the actual structure and size of the gate array to be used. A GRC which contains one or more terminal pins of a particular net is called a *terminal* GRC for that net. Each GRC is indexed by three integers (i, j, k), where i is

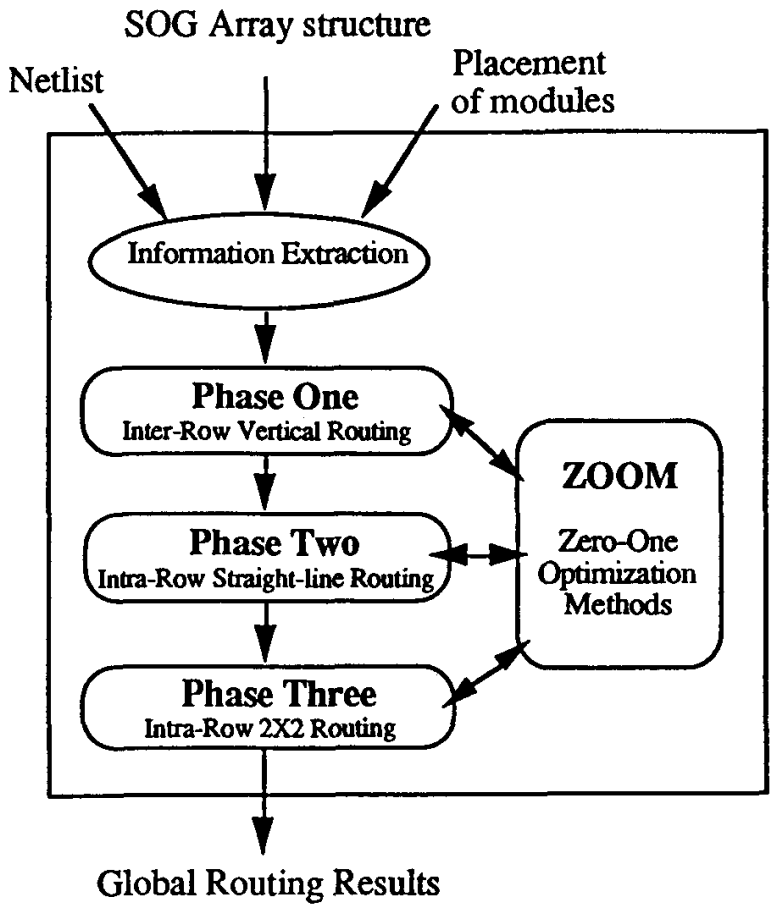


Fig. 10. Overview of SOG router.

the row, j is the column, and k is the subrow of the GRC.

The vertical and horizontal pitch of the GRCs determine the *boundary capacities* and the *via capacities* of the GRCs. These quantities are appropriately reduced once the prerouting in the macrocells is extracted. The boundary capacity is used in the ILP to specify the maximum number of global routes that can cross a GRC boundary, and the via capacity defines the maximum number of global routes that can make a bend in a GRC, since each bend results in a via. As the routing algorithm creates new routes for the various nets, these capacity values are updated accordingly.

Following the information extraction step, global routing is performed using three layers of metal interconnects. For each net, global routes are formed from path segments which traverse a subset of the GRCs of the net. Instead of forming the entire rectilinear Steiner tree at once, the global route for a net is decomposed into vertical and horizontal path segments, which are then routed separately in the three global routing phases until all the pins of the net are connected together.

The routing strategy in all the three phases is similar: the router identifies possible routes for all path segments, and represents each one by a binary variable.

An ILP is formulated using these variables, with the objective of selecting one route for each path segment such that the *benefit* of the routes selected is maximized. The benefit of selecting a route is determined by a number of factors, which will be discussed later. The constraints in the 0-1 ILPs are imposed by track and via availability, and the avoidance of loops in a global route. The three phases, performed in the order shown, are

Phase One (Inter-Row Routing) A vertical straight-line metal-3 path segment is chosen for each net that has terminals in more than one GRC row. This path segment forms a vertical backbone for the completed spanning tree.

Phase Two (Intra-Row Straight Line Routing) This phase consists of two subphases: namely Phase 2V, in which metal-1 vertical path segments are formed between the two subrows of each GRC row, and Phase 2H, in which metal-2 horizontal path segments are formed within each subrow. The metal-2 horizontal path segments are formed as ribs emanating from the metal-3 vertical backbone.

Phase Three (Intra-Row Two-by-Two Routing) In this phase, 2 x 2 cell routing problems are solved hierarchically to obtain path segments that can extend beyond the bounding rectangle of a net and bend around preroutings or blockages.

Figure 11 shows the segments of a net created in each of the phases. The following sections describe each of these three phases in detail.

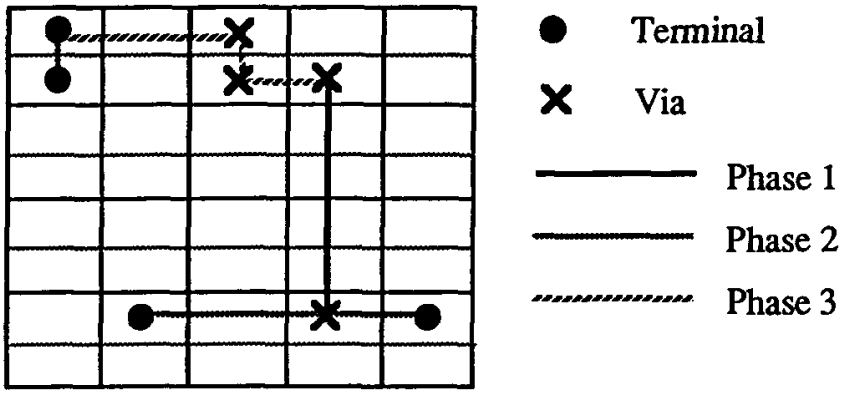


Fig. 11. Segments created for a sample net in three phases.

3.3.1. Phase One (Inter-row routing)

In the first global routing phase, only nets that span two or more rows of the GRC array are considered. For each such net, a vertical path segment in the topmost layer (metal-3) is chosen, which extends the entire vertical span of the net.

The candidates from which the final path segment is chosen are vertical path segments in every column that contains a terminal GRC of the net. Each of these possible paths extends the entire vertical span of the net. If the starting and/or ending GRC of a path segment is not a terminal GRC, then that GRC becomes a

via-GRC since a via will need to be introduced there at a later stage. Figure 12 illustrates the candidate Phase One path segments for a sample net.

The path segments are chosen from the candidates by the Phase One ILP, which is formulated as follows:

MAXIMIZE

$$\sum_{n}\sum_{j} B_{nj} P_{nj}^{m3}$$

SUBJECT TO

Path Usage Constraints: for each net n,

$$\sum_{j} P_{nj}^{m3} \leq 1$$

South Boundary Constraints: for each GRC (i, j, k),

$$\sum_{\text{all } n_{i,j,k}} P_{n_{i,j,k},j}^{m3} \leq S_{i,j,k}^{m3}$$

Via Constraints: for each GRC (i, j, k),

$$\sum_{\text{all } n_{i,j,k}} P_{n_{i,j,k},j}^{m3} \leq V_{i,j,k}$$

0-1 Variable Constraints: for all n and j,

$$P_{n,j}^{m3} = 0 \text{ or } 1$$

where

P_{nj}^{m3} is the path of net n in column j;
S_{ijk}^{m3} is the metal-3 South boundary capacity of the (i, j, k)th GRC;
V_{ijk} is the via capacity of the (i, j, k)th GRC;
$n_{i,j,k}$ refers to nets with possible metal 3 paths crossing the (i, j, k)th GRC,
and $B_{n,j}$ is the benefit of connecting net n using the path in column j.

The benefit $B_{n,j}$ of connecting net n using the vertical path segment in GRC column j is computed heuristically based on the following considerations:

- When competing for the same resource (such as a track or a via), a path belonging to a higher priority net should be given preference over a path belonging to a low priority net.
- A path which traverses more terminal-GRCs should be preferred over one which traverses fewer terminal-GRCs, since the former can be expected to lead to a shorter spanning tree.

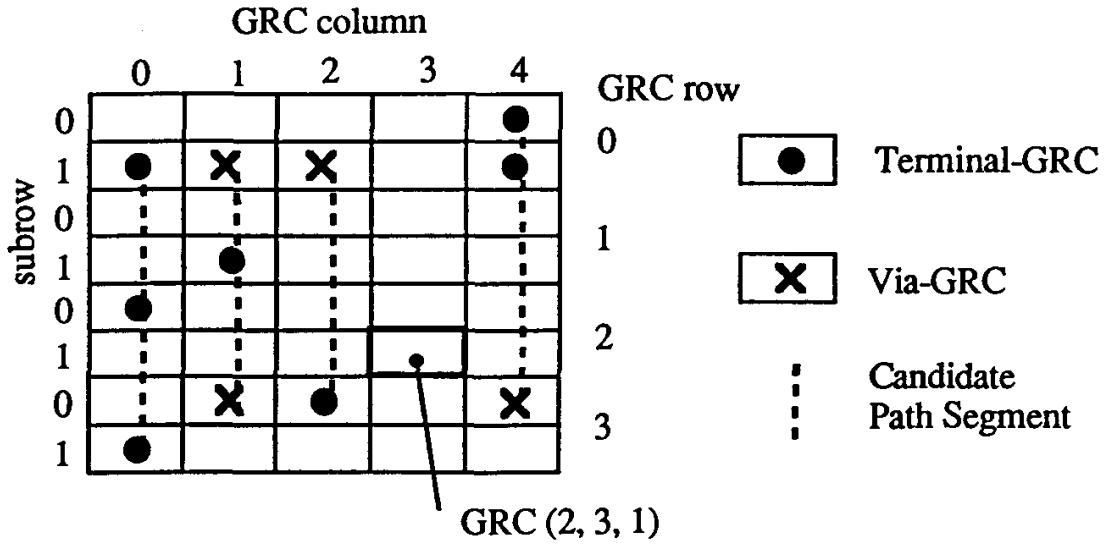


Fig. 12. Candidate Phase One path segments for a sample net.

- A path with fewer via-GRCs uses up fewer resources and should thus be preferred.
- Shorter paths should be used whenever possible.

These considerations translate to the following expression for $B_{n,j}$:

$$B_{n,j} = P_n + T_{n,j} - X_{n,j} - \frac{R_{n,j}}{R_{\text{total}}} + 3$$

where

P_n is the priority value of net $n, 0 \leq P_n \leq \mu$, where $\mu = 10$;
$R_{n,j}$ is the number of GRC rows spanned by the vertical path;
R_{total} is the total number of GRC rows in the routing area;
$X_{n,j}$ is the number of via-GRCs created at each end of the path in column j;
and $T_{n,j}$ is the number of terminal-GRCs of net n in column j.

The constant 3 is added so that the benefit of any segment cannot be less than unity.

The path usage constraints in the ILP allow at most one vertical path to be selected for each net. This restriction helps to prevent the formation of unnecessary loops in the global routes during subsequent phases of the routing. This is illustrated in Fig. 13.

The South boundary constraints ensure that the number of vertical tracks passing through a GRC does not exceed the capacity of the GRC. By introducing a separate constraint for each GRC, the router can allow nonoverlapping paths to share the same tracks, as shown in Fig. 14.

Whenever a Phase One path segment begins or ends in a GRC that is not a terminal GRC for its net, a via is introduced. The via capacity constraints ensure

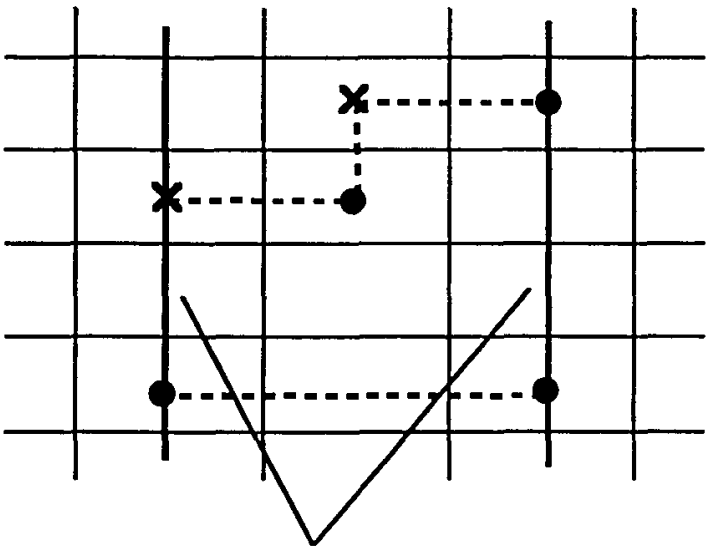


Fig. 13. Phase One path usage constraints.

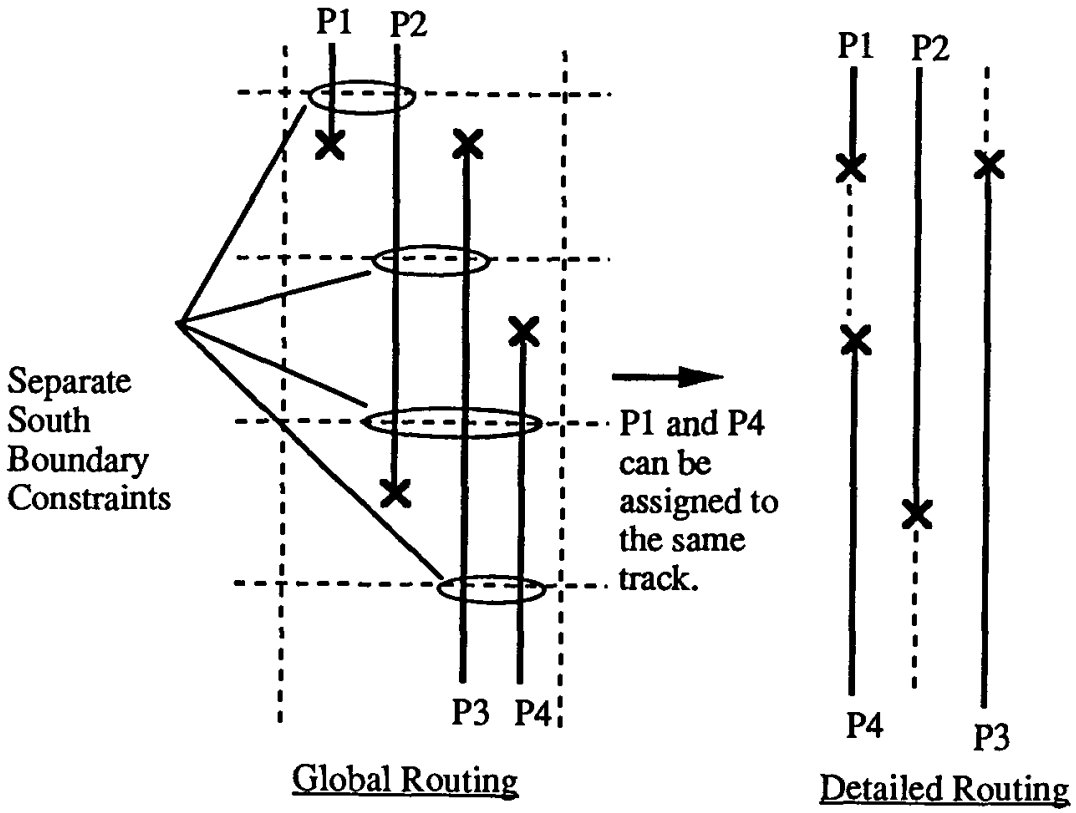


Fig. 14. Phase One south boundary constraints.

that the number of vias introduced in a GRC does not exceed the via capacity of the GRC.

During Phase One, the user can specify how much of the total area should be considered for simultaneous routing. This feature allows the user to break up very large problems into smaller, more manageable subproblems. Through the use of this simple partitioning feature, the global router was capable of handling test cases of over 108,000 transistors and 9,400 nets.

The complexity of the candidate path selection procedure is $O(N_{c'}\ C)$, where $N_{c'}$ is the number of nets that are considered in Phase One, and C is the number of columns. The ILP solution time is $O(nS)$, where n is the number of ILPs to be solved, and S is the size, in terms of the number of constraints and variables, of the ILPs. S is limited by the ILP solver to 100 typically, while n lies between 1 and C.

3.3.2. Phase Two (Intra-row straight line routing)

Since the inter-row interconnections for all nets are taken care of in Phase One, all the interconnections created in Phase Two are straight-line path segments that lie *within* each GRC row. Each row constitutes a separate subproblem. Phase Two routing is divided into two separate subphases, namely Phase 2V (Vertical) and Phase 2H (Horizontal) routing.

Phase 2V (Vertical) routing

In Phase 2V, straight-line vertical metal-1 (topmost layer) path segments are formed between pairs of terminal GRCs lying in the two different subrows of a row. Figure 15 illustrates a net with possible straight line metal-1 paths. For each row r, the 0-1 ILP is formulated so that the router selects as many of the possible path segments as possible:

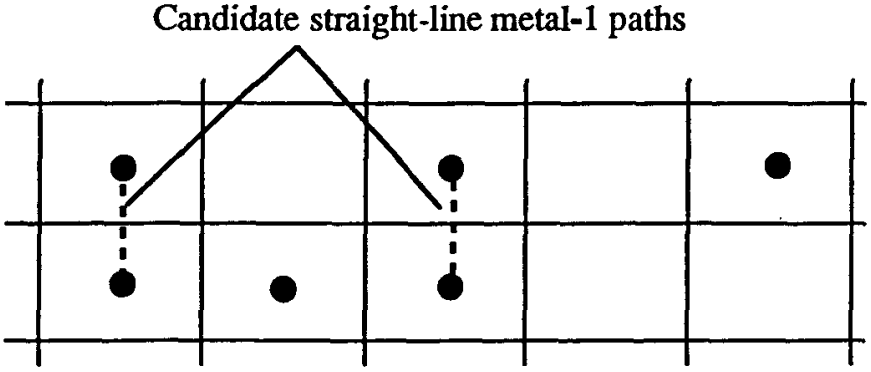


Fig. 15. Phase 2V path segments for a sample net.

MAXIMIZE

$$\sum_{n}\sum_{j} B_{nj} P_{nj}^{m1}$$

SUBJECT TO

South Boundary Constraints: for each column j in row r,

$$\sum_{\text{all } n_j} P_{n_j,j}^{m1} \leq S_{i,j,0}^{m1}$$

0-1 Variable Constraints: for all n and j,

$$P_{n,j}^{m1} = 0 \text{ or } 1$$

where

P_{nj}^{m1} is the path of net n in column j of row r;
S_{rj0}^{m1} is the metal-1 South boundary capacity of the $(r, j, 0)$th GRC;
and $B_{n,j}$ is the *benefit* of connecting net n using the metal-1 path in column j, of row r.

In Phase 2V routing, the benefit $B_{n,j}$ depends only on the priority P_n of the net n and not on the path j, so that all paths for a given net are equally beneficial when selected. A path will be rejected only when it competes for resources with another path belonging to a higher priority net. $B_{n,j}$ is computed as

$$B_{n,j} = P_n + 1$$

The only constraints imposed during Phase 2V routing are the GRC south boundary capacity constraints, since the router tries to maximize the number of path segments selected, and none of the path segments contributes to a via during this phase.

Phase 2H (Horizontal) routing

In Phase 2H, horizontal straight-line metal-2 path segments are formed between pairs of terminal-GRCs, via-GRCs and/or vertical metal-3 paths, as illustrated in Fig. 16. The ILP formulation for Phase 2H in GRC row r is as follows:

MAXIMIZE

$$\sum_{n}\sum_{k}\sum_{j} B_{n,k,j} P_{n,k,j}^{m2}$$

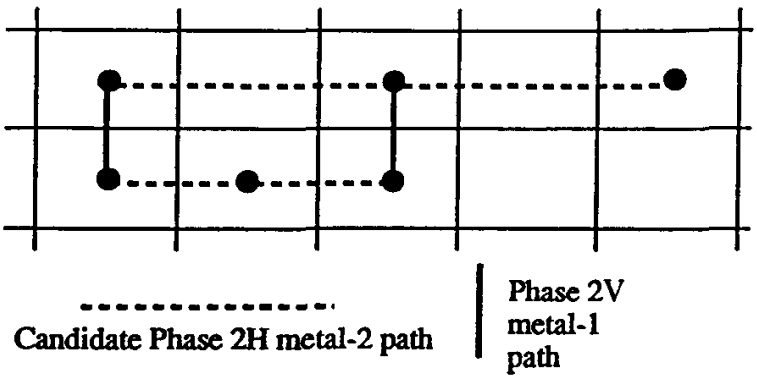


Fig. 16. Phase 2H routing.

SUBJECT TO

East Boundary Constraints: for each j, k, in row r,

$$\sum_{\text{all } n_{r,j,k}} P^{m2}_{n_{r,j,k},j} \leq E^{m2}_{r,j,k}$$

Via Constraints: for each j, k, in row r,

$$\sum_{\text{all } n_{r,j,k}} P^{m2}_{n_{r,j,k},j} \leq V_{r,j,k}$$

Loop Avoidance Constraints: for each loop l,

$$\sum_{\text{all } k} \sum_{\text{all } j_l} P^{m2}_{n_{i,k},j_l} \leq (N_l - 1)$$

0-1 Variable Constraints: for all n, k and j,

$$P^{m2}_{n,k,j} = 0 \text{ or } 1$$

where

P^{m2}_{nkj} is the horizontal metal-2 path of net n in subrow k of row r, starting at column j;

E^{m2}_{rjk} is the metal-2 East boundary capacity of the (r, j, k)th GRC;

V_{rjk} is the via capacity of the (r, j, k)th GRC

N_l is the total number of possible horizontal metal-2 paths for net n in loop l;

and $B_{n,k,j}$ is the *benefit* of connecting net n using the horizontal metal-2 path in subrow k (of row r), starting at column j.

In Phase 2H, $B_{n,k,j}$ is computed heuristically as

$$B_{n,k,j} = P_n + \min\left(\frac{C_{\text{total}}}{C_{n,k,j}}, 10\right)$$

where

P_n is the priority value of net $n, 0 \leq P_n \leq \mu$, where $\mu = 10$;

$C_{n,k,j}$ is the number of GRC columns spanned by the horizontal path;

and C_{total} is the total number of GRC columns in the routing area.

As in previous phases, P_n is added to give preference to path segments of higher priority nets. The second component reflects the preference for shorter paths. Due to its form, the second component cannot have a greater weight than the first one.

The boundary and via constraints in the ILP are similar to those in the earlier phases. The extra loop avoidance constraint introduced in this phase prevents the router from choosing all the N_l possible horizontal paths for a net, since this would lead to the formation of a loop in the global route for the net. Figure 17 illustrates how the loop avoidance constraint works — if all three possible paths for the net are selected, a loop will be formed. But due to the constraint, only two of the paths are selected, resulting in one of the three loopless routes shown.

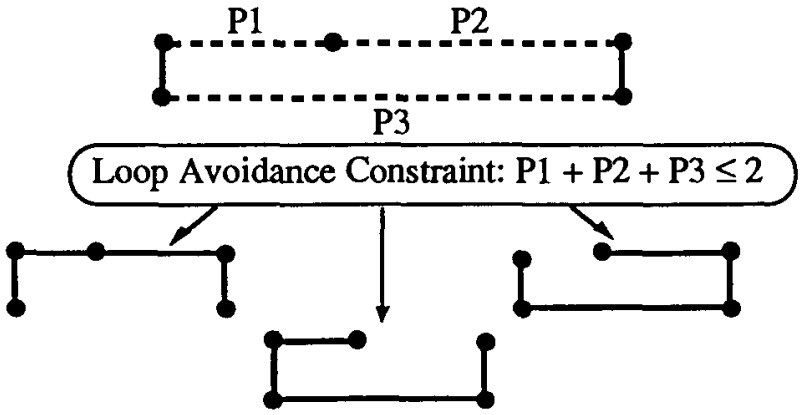


Fig. 17. Phase 2H loop avoidance constraints.

3.3.3. Phase Three (Intra-row two-by-two routing)

In the third and final phase of the global routing, a 2 × N hierarchical approach (N is the number of GRC columns) is used for intra-row horizontal metal-2 interconnects and localized vertical metal-1 interconnects. As in Phase Two routing, all path segments are formed *within* each GRC row. The goal of this routing phase is to complete the routings left over after the first two phases have formed all possible straight-line path segments for the nets. In this phase, the routes constructed may be circuitous, taking as many bends as required to avoid blockages due to preroutings in the macromodules or gates.

Table 2. Differences between SOG router and Burstein and Pelavin's method.

Burstein and Pelavin's Method	SOG Router
Purely 2 x N routing — not preceded by straight-line interconnections.	2 x N routing is preceded by straight-line interconnections in Phase One and Two.
No corrective measure to level out non-uniform boundary capacities.	Phase 2V and 2H serve to level out non-uniform GRC boundaries.
Restrictive routing — a wire route cannot cross a vertical boundary twice.	Non-restrictive — a route may cross a vertical boundary twice.
Fewer routing configurations are considered.	More configurations due to connections made in Phases One and Two.
Strictly top-down hierarchical approach.	Meet-in-the-middle approach.
Determines only the *number* of nets to be used for each configuration.	Determines exactly how each net should be routed.

The routing strategy used in Phase Three is an adaptation and an extension of the 2 x N routing methodology used by Burstein and Pelavin.[39] Each cell in the 2 x 2 cell wiring problems is formed by grouping one or more GRCs in the same GRC subrow, and is referred to as a *Macro-GRC*. Despite the similarities, the hierarchical setup of Phase Three routing is different in some significant ways from the approach of Ref. 39. The key differences are summarized in Table 2.

Phase Three routing is a combination of top-down and bottom-up hierarchical methods, involving four steps that are carried out as laid down by the following algorithm:

```
FOR each row in the routing area, DO {
    Step 1:
        /* Invoke the 2 x 2 macro-GRC routing algorithm at the lowest level L */
        two_by_two_route (L);
    Step 2:
        /* Perform "Meet-in-the-Middle" routing */
        Initialize:
            top_level = 1; bottom_level = L-1; top = FALSE;
        WHILE ( top_level ≤ bottom_level ) DO {
            /* alternate between top-down and bottom-up routing */
            top = NOT (top); /* toggle the top flag */
            IF (top)
                /* do top-down routing */
                /* Invoke the 2 x 2 macro-GRC routing algorithm at the top level */
                two_by_two_route(top_level);
                top_level = top_level + 1;
```

```
            ELSE
                /* do bottom-up routing */
                /* Invoke 2 x 2 macro-GRC routing algorithm at the bottom level */
                two_by_two_route(bottom_level);
                bottom_level = bottom_level - 1;
        }
    Step 3:
        /* perform "downward direction" 2 x 2 macro-GRC routing at each level,
        starting from one level below which top-down routing stopped, until one level
        above the lowest hierarchy */
        FOR level = top_level TO L - 1 {
                two_by_two_route(level);
        }
    Step 4:
        /* perform 2 x 2 macro-GRC routing at the lowest hierarchy */
        two_by_two_route(L);
}
```

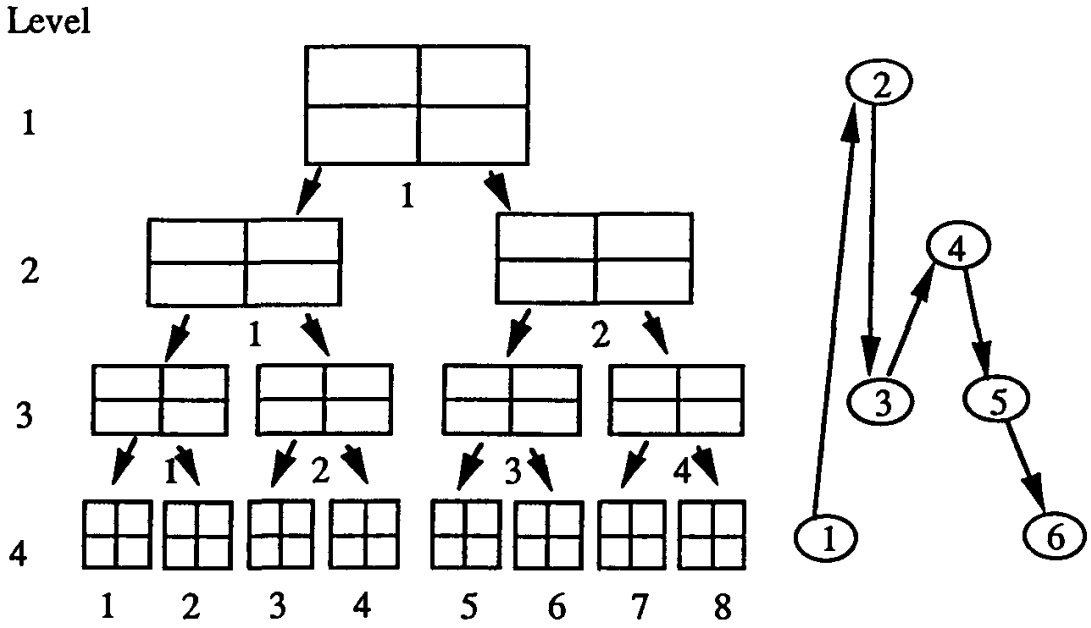


Fig. 18. Hierarchical steps for a 2 x 16 problem.

Figure 18 illustrates the hierarchical steps involved in solving a 2 x 16 problem. The basic unit of the Phase Three routing algorithm is 2 x 2 macro-GRC routing. Before this procedure can be explained, the following definitions need to be made:

- A *macro-node* is defined as a macro-GRC in which at least one of its member GRCs is a terminal GRC, a via-GRC or a *potential* via-GRC, which could be formed in all except the lowest level of the hierarchy. A potential via-GRC is one in which a horizontal path segment crossing a macro-GRC boundary ends, and

the decision to make a horizontal or vertical segment from this GRC has not yet been made.

- A *macro-edge* is formed when there is at least one routed segment crossing the boundary between two macro-GRCs.

The goal of 2 x 2 macro-GRC routing is to classify the partially connected nets, identify all possible macro-edges between the macro-nodes, and select some or all of these edges to achieve an optimal routing. This problem is represented as a graph with four vertices (V) and four edges (E), with one vertex in each macro-GRC and one edge crossing each cutline between two macro-GRCs (Fig. 19). A vertex is labelled black (B) if it is in a macro-node, and white (W) otherwise. Similarly, an edge labelled black (b) represents a macro-edge, and an edge labelled white (w) indicates the absence of a macro-edge.

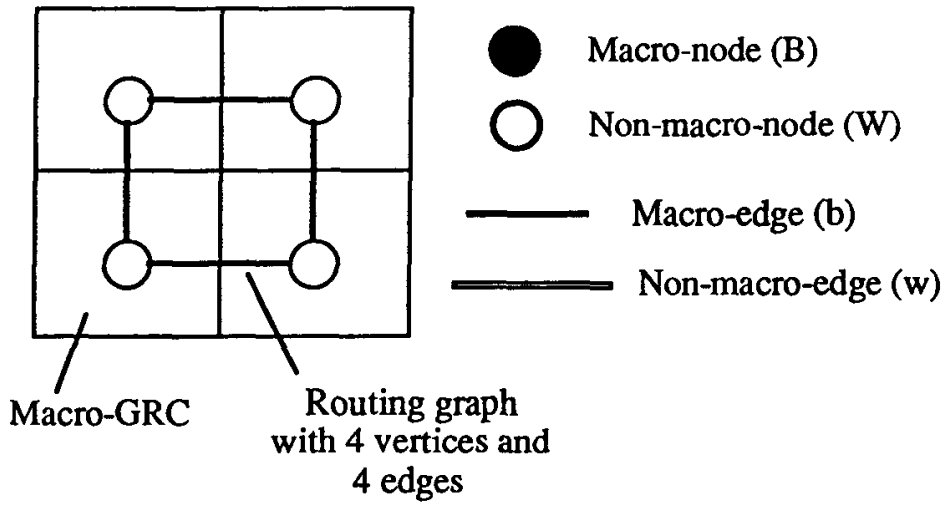


Fig. 19. Macro-GRC 2 x 2 routing.

It can be shown that of the 70 equivalence classes of 2 x 2 net configurations with e macro-edges and n macro-nodes ($e, n = 0, \ldots, 4$), only 29 need to be considered in Phase Three, since the rest correspond to situations which are physically impossible or in which no further routing is required.[31] In the 2x2 macro-GRC routing problem, the nets are classified into these 29 classes. Once the net classification is done, the possible routes for a particular classification are identified from a database, using a fast and simple "table lookup" procedure.

The 0-1 ILP formulated to select the possible routes for the 2 x 2 macro-GRC routing is as follows:

MAXIMIZE

$$\sum_{n} \sum_{c} B_{n,c} R_{n,c}$$

SUBJECT TO

Net Connection Constraints: for each net n,

$$\sum_{c} R_{n,c} \leq 1$$

Boundary Constraints:

$$\textit{for East boundary } E(0), \quad \sum_{\text{all } n_{E(0)}} R_{n_{E(0)},c} \leq E^{m2}_{r,j,0}$$

$$\textit{for East boundary } E(1), \quad \sum_{\text{all } n_{E(1)}} R_{n_{E(1)},c} \leq E^{m2}_{r,j,1}$$

$$\textit{for South boundary } S(0), \quad \sum_{\text{all } n_{S(0)}} R_{n_{S(0)},c} \leq \sigma^{m1}_{S(0)}$$

$$\textit{for South boundary } S(1), \quad \sum_{\text{all } n_{S(1)}} R_{n_{S(1)},c} \leq \sigma^{m1}_{S(1)}$$

Via Constraints: for each macro-GRC (x, y),

$$\sum_{\text{all } n_{(x,y)}} R_{n_{(x,y)},c} \leq \vartheta_{(x,y)}$$

0-1 Variable Constraints: for all n and c,

$$R_{n,c} = 0 \text{ or } 1$$

where

$R_{n,c}$ is the 0-1 variable corresponding to route c for net n;
E^{m2}_{rjk} is the metal-2 East boundary capacity of the (r, j, k)th GRC, which is the rightmost GRC constituting the macro-GRC $(k, 0)$;
$\sigma^{m1}_{S(k)}$ is the average value of the metal-1 South boundary capacity of all the GRCs constituting the macro-GRC $(0, k)$;
$\vartheta_{(x,y)}$ is the average via capacity of all the GRCs constituting the macro-GRC (x, y);
and $B_{n,c}$ is the *benefit* of connecting net n using the choice c route.

The benefit $B_{n,c}$ of connecting net n using the choice c route in the 2×2 routing problem is calculated using the following expression:

$$B_{n,c} = P_n + (6 - F_{n,c})$$

where

P_n is the priority value of net n, $0 \leq P_n \leq \mu$, where $\mu = 10$;

and $F_{n,c}$ is the total number of possible vias and path segments added if choice c is used to connect net n, $1 \leq F_{n,c} \leq 5$.

This expression favors routes of higher priority nets, as before, and the second component favors routes requiring fewer path segments and vias. Since all the route configurations give rise to at most five path segments and vias, the constant 6 is included so that the values of the second component of the benefit range from 1 to 5, and the minimum benefit is 1.

In Phase Three routing, the boundary capacities E(0) and E(1) are *actual* GRC boundary capacities, because the horizontal path segments actually cross the east boundaries of the GRCs. However, the vertical track availability in S(0) and S(1) is computed from the *average* of all the south boundary capacities of the GRCs constituting the macro-GRC $(0, k)$, since the assignment of vertical track segments to particular GRCs is not done until the lowest level of the hierarchy is reached.

3.4. Results for Medium Size Arrays

The SOG router program has run successfully on real industrial circuits. Results on a CMOS circuit c162, and a BiCMOS circuit c1318, are presented in this section. Table 3 gives the statistics of these circuits, including information on the placement and packing densities. The router was run on these circuits with varying GRC pitch sizes. The results of the global routing are shown in Table 4. In all the cases, all of the nets were completely routed within the layout area, thus achieving 100% "over-the-cell" global routing. The wireable packing densities of the circuits compare very favorably with those of many SOG chips reported in the literature. Figure 20 compares the packing density of c162 (normalized at different feature sizes of the process technology) with CMOS chips from Fujitsu,[40] Hitachi,[41] Mitsubishi,[42] Philips [43] and Stanford University.[44] The packing density of c1318 is compared with BiCMOS chips from Fujitsu,[45] NTT LSI laboratories,[46] LSI Logic Corporation,[47] and Mitsubishi[48] in Fig. 21.

Table 3. Type, size and placement information for c162 and c1318.

Name of Circuit	c162	c1318
Type of Circuit	CMOS	BiCMOS
No. of Transistors	450	4657
No. of Modules	116	956
No. of Nets	162	980
Layout/Routing Area	34,145 μm^2	724,992 μm^2
Packing Density	75.9 μm^2/trans.	155.7 μm^2/trans.

Table 4. Global routing statistics for c162 and c1318.

Name of Circuit	c162		c1318		
No. of GRC rows	4	4	8	8	8
No. of GRC columns	8	16	8	16	32
P^h	7	4	73	36	18
P^v	9	9	9	9	9
% of nets routed	100%	100%	100%	100%	100%
No. of ILPs (Actual/Max)					
Phase One	1/8	2/16	8/8	8/16	16/32
Phase Two	5/8	5/8	16/16	16/16	16/16
Phase Three	22/52	25/108	72/104	127/216	379/472
Total	28/68	32/132	96/128	151/248	411/520
Convex CPU time	9.19 s	16.00 s	105.88 s	204.13 s	733.05 s

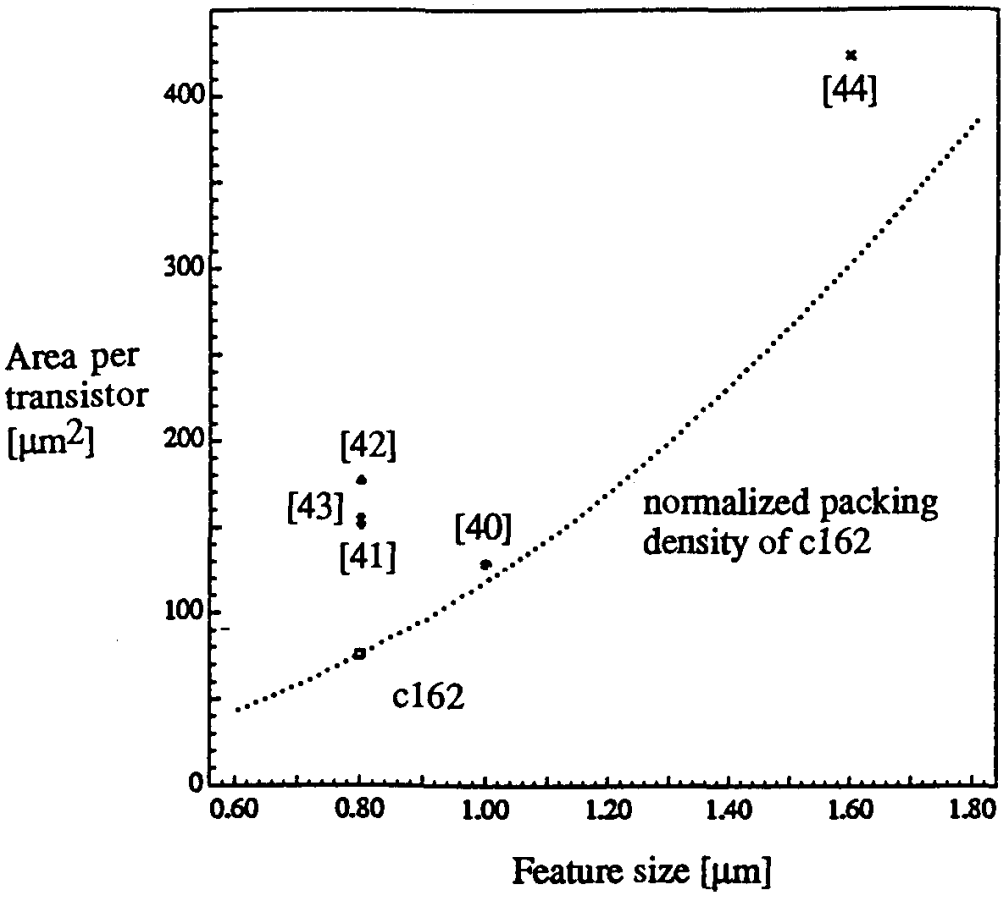


Fig. 20. Comparison of packing densities of c162 and some industrial CMOS chips.

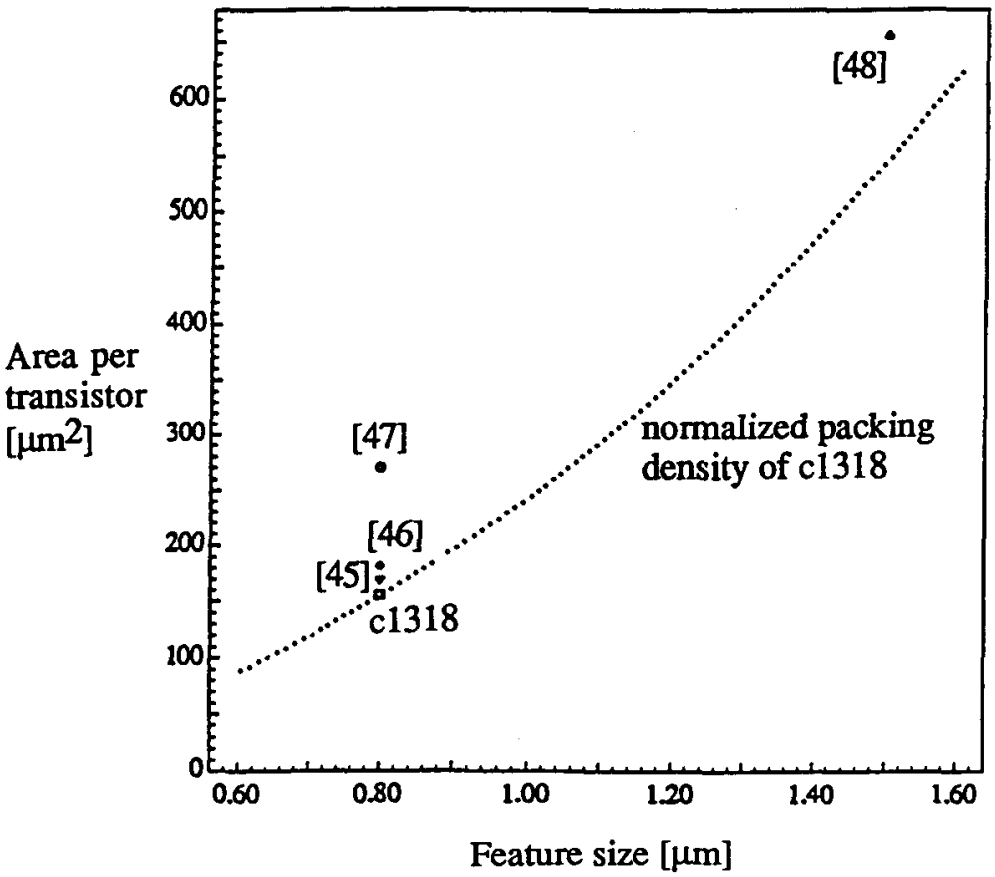


Fig. 21. Comparison of packing densities of c1318 and some industrial BiCMOS chips.

3.5. Application to Custom Logic Layout

The ILP approach can also be extended to custom logic module design, to generate compact layout for logic blocks of about 1000 transistors. The system can combine a standard cell approach to placement with a sea-of-gates array layout to perform the routing and transistor-level placement. The size of each cell will be an estimate based on the number of transistors within the cell. Once the estimation is done, a standard cell placement program can be used to perform the placement of the logic cells. This placement can then be mapped onto a SOG array, and the global router can then determine which GRCs each net will pass through. After the global routing phase, the signal delays for each net can be estimated, and these values can be used to optimize the transistor sizes for better performance. The data from all these stages can then be fed to a fine routing and placement program. The data from the global routing initially serves to create a set of loose, or unassigned, boundary constraints. As the program proceeds to process GRCs, some of the boundary constraints change from being loose to tight, or restricted, constraints. The process can be repeated until the chip design is completed. Figure 22 shows

the global routing result for a four bit adder circuit with 144 transistors and 45 nets, obtained using the above approach.

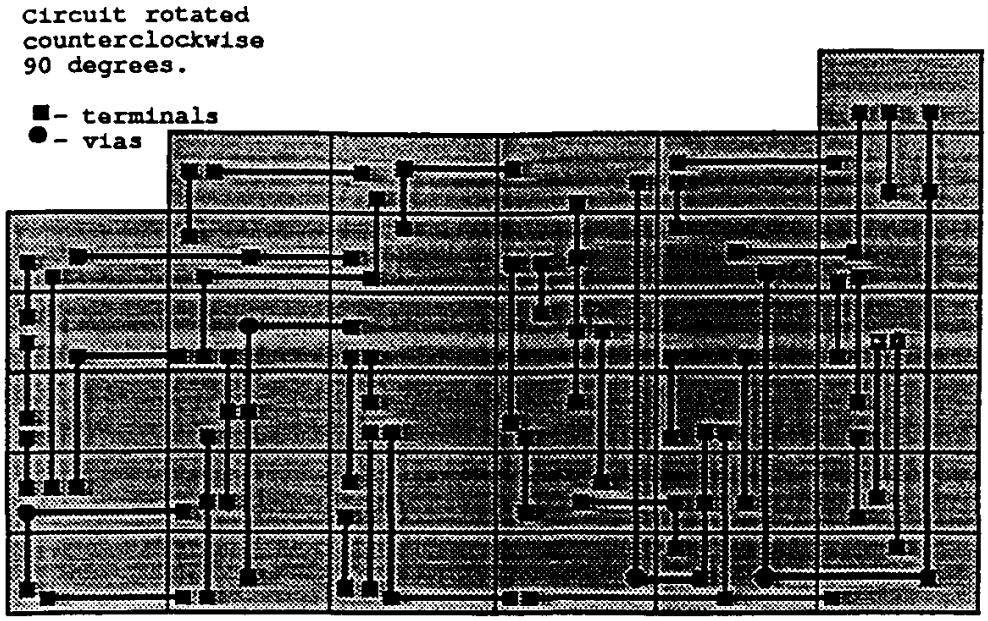


Fig. 22. Global routing of a 4-bit adder circuit.

3.6. Results on Very Large Circuits

Until now, linear programming approaches for global routing have been severely limited by the maximum size of the ILP that can be handled by them. However, due to the novel approach taken in the new SOG router, it is capable of handling much larger circuits than previous approaches. To demonstrate this capability, the router was run on an industrial BiCMOS circuit with over 27,000 transistors. The program was able to route all 2396 nets within the channelless layout area with a CPU runtime of only 821 seconds. The packing density achieved in about 0.8 μm technology was very high, 151.3 μm^2 per transistor.

In another test run, two copies of the 27,000 transistor circuit were placed side-by-side, and connections between the two circuits were formed to obtain a circuit with 54,000 circuits and 4794 nets. Once again, the router was able to achieve a 100% routing within the allotted area. Using this approach to obtain large examples, the router was run successfully on circuits with up to 108,000 transistors. To the best of the authors' knowledge, routing of such large circuits using ILPs has not been reported. The maximum size of circuits that can be handled by the router is limited by the size of the ILP that can be handled by the ILP solver. This limitation can be overcome using an algorithm for circuit partitioning with interface pin assignment.[49]

3.7. Run Time Complexity

Time complexity tests for the global router were performed by duplicating the c1318 circuit with 4657 transistors and 980 nets, to form test cases having 2, 4, 6, and 9 times the original number of transistors. The cpu-times for the five examples are plotted in Fig. 23. As the data shows, the cpu-time increases approximately linearly with the problem size. The factor that influences the execution time the most is the number of ILPs that are generated and solved, and this number in turn depends on the local density of nets, the number of rows and columns in each partition, the size of the overall circuit, and the problem size that the ILP solver can handle.

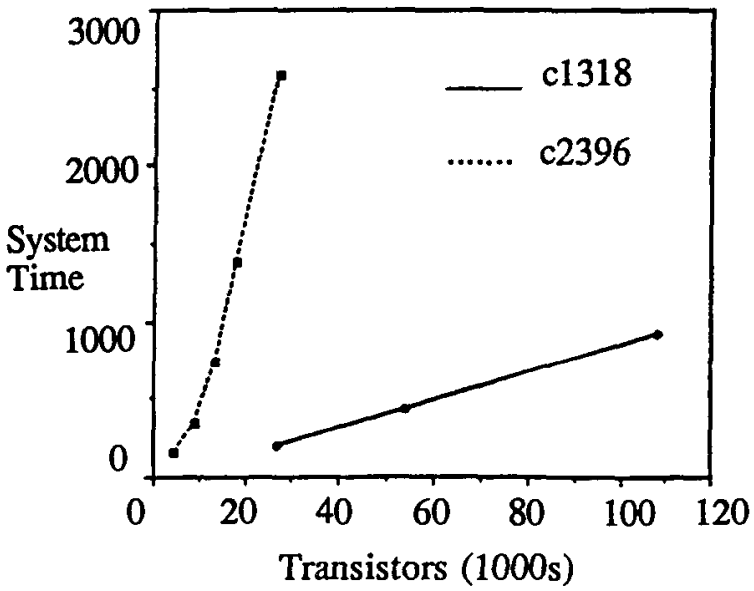


Fig. 23. Plot of cpu-time versus problem size.

The linear run time behavior was found to hold even for very large circuits, which were generated by duplicating the 27,000 transistor circuit c2396 in the manner described above. A point worth noting is that the artificial circuits created by duplication provide an over-estimate of the required run time for actual circuits having the same number of transistors. As the figure shows, the run time needed for the six-times duplicated circuit c1318 was much greater than the run time for a single c2396 circuit, even though the number of transistors is approximately the same for both. This is mainly due to the fact that the net count in the replicated circuit is higher than that in the circuit with the same number of transistors.

4. Summary

In this chapter, binary formulations in terms of integer variables that assume binary values have been developed for module placement and global routing problems. These formulations have been shown to be amenable to rigorous optimiza-

tion techniques such as artificial neural networks or 0-1 integer linear programming techniques.

In Section 2, the two-dimensional module placement problem was mapped onto a modified Hopfield neural network consisting of two coupled one-dimensional Hopfield networks. The output of individual neurons is either $+1$ or -1. To avoid local minima, simulated annealing techniques have been introduced in the neural network. The performance of the network has been simulated with a computer program using some benchmark examples. Despite the limitation of Hopfield network architecture, the solutions provided by the neural network come very close to those of a high quality industrial placement program. The results encourage further investigation along this direction, especially for potential hardware implementation of this placement technique.

The sea-of-gates design style provides not only fast turnaround time, but also better speed performance in alliance with bipolar transistors in case of BiCMOS technology. Global routing of sea-of-gates is an important physical design issue. In Section 3, the three-layer global routing problem has been formulated in terms of a 0-1 integer linear programming problem and solved by ZOOM, a powerful optimization program. As demonstrated through some benchmark examples, the quality of solutions obtained by this approach matches or even exceeds the published state-of-the-art results with full automatic routing.

It is believed that simple binary techniques can also be applied to other physical VLSI design problems such as quantized transistor sizing as in the case of standard-cell based circuit optimization.

References

1. J. Soukup, "Circuit placement", *Proceedings of the IEEE*, **69**, pp. 1281–1304, 1981.
2. S. Kirkpatrick, C. D. Gelatt Jr. and M. P. Vecchi, "Optimization by Simulated Annealing", *Science*, **220**, pp. 671–680, 1983.
3. C. Sechen and A. Sangiovanni-Vincentelli, "The TimberWolf placement and routing package", *Proceedings of the Custom Integrated Circuits Conference*, May 1984.
4. R. M. Kling and P. Banerjee, "ESP: Placement by Simulated Evolution", *IEEE Trans. CAD*, **8(3)**, pp. 245–256, March 1989.
5. Y. G. Saab, "Combinatorial optimization by Stochastic Evolution with applications to the physical design of VLSI circuits", Doctoral Thesis, University of Illinois at Urbana-Champaign, 1990.
6. M. A. Breuer, "Min-cut placement", *J. Design Automation and Fault Tolerant Computing*, **I(4)**, pp. 343–362, Oct. 1977.
7. A. E. Dunlop and B. W. Kernighan, "A procedure for placement of standard cell VLSI circuits", *IEEE Trans. on Computer-Aided Design*, **CAD-4(1)**, pp. 92–98, Jan. 1985.
8. P. R. Suaris and G. Kedem, "An algorithm for quadrisection and its application to standard cell placement", *IEEE Transactions on Circuits and Systems*, **35(3)**, pp. 294–303, March 1988.
9. R. P. Lippmann, "An introduction to computing with neural nets", *IEEE ASSP Magazine*, pp. 4–22, April 1987.

10. J. J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities", *Proceedings of the National Academy of Science, USA*, **79**, pp. 2554–2558, 1982.
11. J. J. Hopfield, "Neurons with graded response have collective computation properties like those of two-state neurons", *Proc. Natl. Acad. Sci.*, **81**, pp. 3088–92, May 1984.
12. J. J. Hopfield and D. W. Tank, "Neural" computation of decisions in optimization problems", *Biological Cybernetics*, **52**, pp. 141–152, 1985.
13. D. W. Tank and J. J. Hopfield, "Simple 'neural' optimization networks: An A/D converter, signal decision circuit, and a linear programming circuit", *IEEE Transactions on Circuits and Systems*, **33(5)**, pp. 533–541, May 1986.
14. M.-L. Yu, "A study of the applicability of Hopfield decision neural nets to VLSI CAD", *Proc. 26th ACM/IEEE Design Automation Conference*, pp. 412–417, 1989.
15. B. C. Levy and M. B. Adams, "Global optimization with stochastic neural networks", *IEEE First Annual International Conference on Neural Networks*, June 1987.
16. E. Wong, "Stochastic neural networks", *Memorandum No. UCB/ERL M89/9*, Electronics Research Laboratory, University of California at Berkeley, Feb. 1989.
17. J. H. Cervantes and R. R. Hildebrant, "Comparison of three neuron-based computation schemes", *Proc. IEEE First Annual International Conference on Neural Networks*, pp. III-657-671, June 1987.
18. C. Peterson and J. R. Anderson, "Neural networks and NP-complete optimization problems: A performance study on the Graph Bisection problem", *Complex Systems*, **(2)**, pp. 59–89, 1988.
19. W. Swartz and C. Sechen, "New algorithms for the placement and routing of macro cells", *Proc. IEEE International Conference on Computer-Aided Design*, 1990.
20. C. Sechen, *VLSI Placement and Global Routing Using Simulated Annealing*, Boston, MA: Kluwer Academic Publishers, 1988.
21. C. Mead, *Analog VLSI and Neural Systems*, Addison-Wesley, 1989.
22. F. J. Kub, K. K. Moon, I. A. Mack and F. M. Long, "Programmable analog vector-matrix multipliers", *IEEE Journal of Solid-State Circuits*, **25(1)**, February 1990.
23. M. J. S. Smith and C. L. Portmann, "Practical design and analysis of a simple 'neural' optimization circuit", *IEEE Transactions on Circuits and Systems*, 36(1), pp. 42–50, January 1989.
24. J. Alspector, R. B. Allen, V. Hu and S. Satyanarayana, "Stochastic learning networks and their electronic implementation", *Proc. Conf. Neural Information Processing Systems*, Denver, Nov. 1987.
25. F. J. Kub, I. A. Mack, K. K. Moon, C. T. Yao and J. A. Modolo, "Programmable analog synapses for microelectronic neural networks using a hybrid digital-analog approach", *IEEE International Conference on Neural Networks*, July 1988.
26. A. Rodríguez-Vázquez, R. Domínguez-Castro, A. Rueda, J. L. Huertas and E. Sánchez-Sinencio, "Nonlinear switched capacitor 'neural' networks for optimization problems", *IEEE Transactions on Circuits and Systems*, **37(3)**, pp. 384–397, March 1990.
27. J.-S. Yih and P. Mazumder, "A neural network design for circuit partitioning", *Proc. 26th ACM/IEEE Design Automation Conference*, pp. 406–411, 1989.
28. R. Libeskind-Hadas and C. L. Liu, "Solutions to the module orientation and rotation problems by neural computation networks", *Proceedings of the 26th ACM/IEEE Design Automation Conference*, pp. 400–405, 1989.
29. P. H. Shih, K. E. Chang and W. S. Feng, "A neural computation network for global routing", in a private communication, 1990.
30. P. Duchene and M. J. Declercq, "A highly flexible Sea-of-Gates structure for digital and analog applications", *IEEE Journal of Solid-State Circuits*, **24(3)**, pp. 576–584, June 1989.

31. Ngee Lek, "A global routing scheme for Sea-of-Gates arrays using zero-one optimization methods", M.S. Thesis, University of Illinois at Urbana-Champaign, 1990.
32. J. P. Cohoon, D. S. Richards and J. S. Salowe, "An optimal Steiner tree algorithm for a net whose terminals lie on the perimeter of a rectangle", *IEEE Trans. CAD*, **9(4)**, pp. 398–407, April 1990.
33. B. Kolman and R. E. Beck, *Elementary Linear Programming with Applications*, New York, NY: Academic Press, Inc., 1980.
34. E. Balas and C. H. Martin, "Pivot and complement — a heuristic for 0-1 programming", *Management Science*, **26(1)**, pp. 86–96, Jan. 1980.
35. J. Singhal, R. E. Marsten and T. L. Morin, "Fixed order branch and bound methods for mixed-integer programming: The ZOOM system", *ORSA Journal on Computing*, pp. 44–51, Winter 1989.
36. R. E. Marsten, "The design of the XMP linear programming library", *ACM Trans. Mathematical Software*, pp. 481–497, Dec. 1981.
37. T. C. Hu and M. T. Shing, "A decomposition algorithm for circuit routing", in *VLSI Circuit Layout: Theory and Design*, New York, NY: IEEE Press, 1985.
38. A. P. Ng, P. Raghavan and C. D. Thompson, "Experimental results for a linear program global router", *Computers and Artificial Intelligence*, pp. 229–242, 1987.
39. M. Burstein and R. Pelavin, "Hierarchical wire routing", *IEEE Trans.on Computer-Aided Design of Circuits and Systems*, **CAD-2**, pp. 223–234, Oct. 1983.
40. Y. Suehiro *et al.*, "A 120K gate usable CMOS Sea-of-Gates packing 1.3M transistors", *Proc. IEEE Custom Integrated Circuits Conference*, pp. 20.5.1–20.5.4, 1988.
41. T. Takahashi *et. al*, "A 1.4M-transistor CMOS gate array with 4nS RAM", *Proc. IEEE International Solid-State Circuits Conference*, pp. 178–179, 1989.
42. M. Okabe *et al.*, "A CMOS Sea-of-Gates array with continuous track allocations", *Proc. IEEE International Solid-State Circuits Conference*, pp. 180–181, 1989.
43. H. Veendrick, D. van der Elshout, D. Harberts and T. Brand, "An efficient and flexible architecture for gate arrays", *Proc. IEEE International Solid-State Circuits Conference*, pp. 86–87, 1990.
44. J. A. Gasbarro and M. A. Horowitz, "A single-chip, functional tester for VLSI circuits", *Proc. IEEE International Solid-State Circuits Conference*, pp. 84–85, 1990.
45. Y. Enomoto, T. Sasaki, S. Tsutsumi and S. Tone, "A 200K gate 0.8 mm mixed CMOS/BiCMOS Sea-of-Gates", *Proc. IEEE International Solid-State Circuits Conference*, pp. 92–93, 1990.
46. H. Yoshimura *et al.*, "500K transistor custom BiCMOS LSI using automated macrocell design", *Proc. IEEE International Solid -State Circuits Conference*, pp. 122–123, 1989.
47 A. Wong *et al.*, "A high-density BiCMOS direct drive array", *Proc. IEEE Custom Integrated Circuits Conference*, pp. 20.6.1–20.6.3, 1988.
48. T. Hanibuchi *et al.*, "A bipolar-PMOS merged basic cell for 0.8μm BiCMOS Sea-of-Gates", *Proc. IEEE Custom Integrated Circuits Conference*, pp. 4.2.1–4.2.4, 1990.
49. T. Parng and R. Tsay, "A new approach to Sea-of-Gates global routing", *Proc. 1989 ICCAD*, pp. 52–55, 1989.

Algorithmic Aspects of VLSI Layout
Eds. M. Sarrafzadeh and D. T. Lee
© World Scientific Publishing Company

A SURVEY OF PARALLEL ALGORITHMS FOR VLSI CELL PLACEMENT*

PRITHVIRAJ BANERJEE
Electrical and Computer Engineering and Coordinated Science Laboratory,
University of Illinois at Urbana-Champaign,
1308 W. Main St., Urbana, IL-61801, USA
(217) 333-6564
banerjee@crhc.uiuc.edu

ABSTRACT

The VLSI cell placement problem is known to be NP complete. A large number of heuristic approaches exist in the literature to solve the problem, which vary in their runtime requirements and layout quality produced. Unfortunately, approaches that produce extremely good quality of layout always take large amounts of computation time. Hence, in recent years, researchers have turned to parallel processing as a viable approach to achieve the tremendous performance improvements that will be needed to design future generations of VLSI chips.

This paper provides a review of parallel algorithms for the VLSI cell placement problem on both shared memory and distributed memory multiprocessors. Six different classes of parallel algorithms for placement are discussed based on the following methods (1) Pairwise interchange (2) Simulated annealing (3) Simulated evolution (4) Genetic approaches (5) Hierarchical decomposition techniques (6) Combination of above approaches.

Keywords: Parallel algorithms, shared memory algorithms, message-passing algorithms, simulated annealing, genetic algorithms hierarchical place and route, iterative improvement.

1. Introduction

In view of the increasing complexity of VLSI circuits, there is a growing need for sophisticated computer-aided design (CAD) tools to automate the synthesis, analysis and verification steps in the design of VLSI systems. Future CAD tools have to enable designs that are too large or complex to undertake otherwise, shorten design time, improve product quality, and reduce product costs.

Almost all the VLSI CAD applications in synthesis, analysis and verification take large runtimes on existing sequential computers. Numerous researchers have

*This research was supported in part by the National Science Foundation under Grant NSF MIP 86-57563 PYI, in part by the National Aeronautics and Space Administration under Grant NASA ICLASS NAG 1-613, and in part by the Semiconductor Research Corporation under Contract SRC 91-DP-109.

started investigating the use of parallel processing for VLSI CAD applications for the following reasons:

(1) Faster runtimes.
(2) Ability to handle larger problem sizes.
(3) Ability to produce better quality of results.
(4) Parallel processing is an affordable and available technology.

In this paper, we will review parallel algorithms for the VLSI cell placement problem, which is known to be NP complete. A large number of sequential heuristic approaches exist in the literature to solve the problem, which vary in their runtime requirements and layout qualities produced. Unfortunately, approaches that produce extremely good quality of layout always take large amounts of computation time. Hence, in recent years, researchers have turned to parallel processing as a viable approach to achieve the tremendous performance improvements that will be needed to design future generations of VLSI chips. Since there has been a large number of sequential heuristic approaches to solve the placement problem, parallel algorithms for this problem are typically based on one or more of these sequential heuristics. Hence, there has been a lot of work reported in the area of parallel placement algorithms, all of which is difficult to cover in detail in a single paper. Therefore this paper should be viewed as an introduction to the field; the reader is encouraged to read the references for more detail.

1.1. Parallel Processing Options for VLSI CAD Applications

With respect to VLSI CAD applications, the parallel processing options that exist are:

(a) *Use of network of workstations.* Almost every CAD environment has some form of networked configuration of workstations, not all of which are completely used all the time. This is perhaps the least costly approach to the use of parallel processing.
(b) *Use of shared memory MIMD multiprocessors.* Currently there are a number of medium priced bus-based shared memory multiprocessors available commercially today. Examples of such machines are the Sequent Symmetry, the Encore Multimax. The programming of these machines can be easily done by parallelizing sequential programs either manually or by parallelizing compilers. Shared memory multiprocessor based workstations have been introduced as well from manufacturers such as Solbourne, Silicon Graphics, SUN, Xerox and Digital.
(c) *Use of distributed memory MIMD multiprocessors.* There are a number of medium-priced hypercube and mesh based multiprocessors commercially available today. Examples are Intel iPSC and i860, NCUBE2, Meiko, and Floating Point Systems. The advantage of these architectures is that they are readily scalable to large number of processors without changing the technology of the interconnect compared to shared memory multiprocessors. However, the pro-

gramming of these distributed memory machines using message passing models is quite difficult. Message-passing multiprocessor based workstations have been introduced by some companies as well such as EEC-1.

(d) *Use of Massively parallel SIMD multiprocessors.* Recently, many researchers have started using massively parallel SIMD multiprocessors such as the Connection Machine for developing parallel CAD applications. However, such machines are rather expensive and are quite difficult to program.

(e) *Use of special purpose hardware accelerators.* Even though such accelerators have the capability to deliver the highest performance, such an approach is not cost effective except for certain applications.

1.2. Placement Problem

In order to understand the placement problem, one has to be familiar with the various automated layout design styles that exist. They include: *standard cell, gate array, sea-of-gates* and *macro cell* or *building block* design styles. Different placement algorithms have been proposed for each of these layout styles. We will first review some of the design styles and then discuss the important placement algorithms.

In the standard cell design style, cells corresponding to frequently used logic modules are predesigned and stored in libraries. These cells have fixed height but varying widths depending on the module functionality and driving capabilities. These cells are laid out in rows, with routing channels reserved between rows of cells for the interconnects. Power and ground lines run horizontally through the cells, and the logic inputs and outputs of the cells are brought out at terminals along the top or bottom edge or both. Connections between the terminals of cells are made by running interconnects through the routing channels. Connections from one row to another is done by using feedthrough cells which are standard cells height cells with a few vertical interconnects. A typical standard cell layout is shown in Fig. 1.

In the gate array design style, the circuit again consists of predefined libraries of modules, except that the most of the mask layers are prefabricated. Specifically, the layout is comprised of a two-dimensional array of *basic cells*, all of equal size, laid out in rows which are separated by routing areas called channels. Basic cells contain isolated transistors and must be "programmed" with connections in different layers of metal. By programming and connecting one or more basic cells together, all of the basic logic gates (i.e., AND, NOR, NOT, ...) and flip-flops can be created. In order to reduce the fabrication time and cost per new design, wafers of gate array chips are fabricated in large amounts until the point of programming and connecting the basic cells is reached. This means that the locations of the basic cells and the height of the channels are fixed. Each new design will then require only a few fabrication steps on the pre-fabricated wafers, and can be completed in much less time. Although the fabrication time is much less, there are some drawbacks

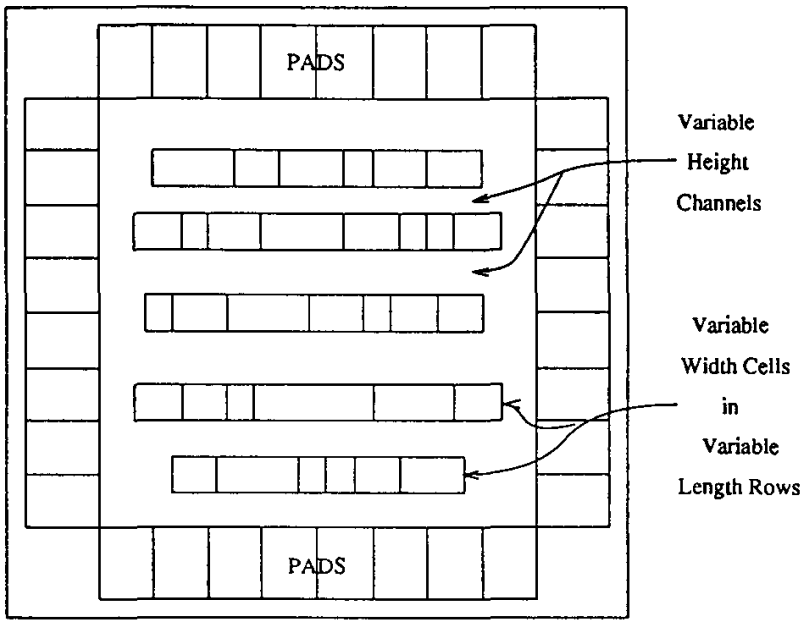


Fig. 1. Example of a standard cell design.

to gate array layout. There is an absolute upper bound on the number of basic cells available and so the number of possible gates is limited. Also, the fixed size of the channels can either restrict the routing of nets or cause much wasted chip area. Often, basic cell utilization is much less than 100%.

Sea-of-gates designs are very similar to gate arrays. The primary difference is that there are no predefined areas for routing. Instead, it is assumed that an extra layer of metal can be used for over-the-cell connections. The number of basic cells is much higher than that of gate array, but the fabrication is more costly since more metal layers are necessary and the layout is more difficult.

In macro block or building block placement, macro cells corresponding to larger predesigned functional units such as memory arrays, decoder, ALUs, are placed on a VLSI chip. The macro cells can be of irregular shapes and sizes and do not necessarily fit together in regular rows or columns. Space is left around the modules for wiring.

Automated layout consists of two primary functions: determining the positions of the modules (standard cells, gates, macro cells) on the VLSI chip, called placement, and interconnecting the modules with wiring, called routing. Although placement and routing are intimately related and interdependent, they have been traditionally separately solved because of their computational complexities. Hence placement algorithms use some crude measures to estimate the routing complexities of various placement, which are refined during the actual routing steps.

The cell placement problem involves placing a set of cells on a VLSI layout, given a netlist which provides the connectivity between each cell and a library containing layout information for each type of cell. This layout information includes the width

and height of the cell, the location of each pin, the presence of equivalent (internally connected) pins, and the possible presence of feedthrough paths within the cell. The primary goal of cell placement is to determine the best location of each cell so as to minimize the total area of the layout and the length of the nets connecting the cells together. One needs to minimize the chip area in order to fit more functionality in a chip. One needs to minimize wirelength in order to reduce capacitive delays associated with longer nets and speed up the operation of the chip. The goals are related since the area of a chip layout is the area of the modules plus the area of the interconnect, hence if the wirelength is reduced the area of the layout is minimized as well.

1.3. Placement Algorithms

Placement methods can be divided into two classes [1–3]: *Constructive and Iterative.* Constructive methods produce a complete placement (all cells have assigned positions) based on a partial placement (some or all cells do not have assigned positions). Constructive algorithms are typically divided into the following classes: (1) Cluster growth [2], (2) Partitioning based placement [4–6], (3) Global techniques such as quadratic assignment and convex function optimization [7], and (4) Branch-and-bound techniques [2].

Iterative methods attempt to improve a complete placement by performing heuristic optimizations. Constructive placement algorithms are normally used for initial placement and are usually followed by iterative algorithms. Within one iteration, certain cells are selected and moved to alternate locations. If the resulting configuration is better than the old one, the new configuration is retained; otherwise, the previous configuration is restored. Some of the conventional iterative improvement techniques include pairwise interchange [8], force-directed interchange, force-directed relaxation [9], and successive overrelaxation methods [10, 11].

Most of the above iterative techniques accept trial placements only if the objective function does not increase. This characteristic may cause an algorithm to get stuck in a local minimum rather than finding the global optimum. The Simulated Annealing technique proposed by Kirkpatrick *et al.* [12] is a general combinatorial optimization technique that uses a probabilistic hill-climbing method to get around this problem. However, this algorithm has several drawbacks, mainly its large CPU time requirements and the need for an efficient annealing schedule [13]. Simulated Annealing has been proven to converge to a globally optimal result given an arbitrary amount of computation time. If less CPU time is provided, it will produce near-optimal solutions. The Simulated Annealing technique has been successfully used in the standard cell placement problem in the TimberWolf placement and routing package [14–16]. Other implementations of Simulated Annealing applied to standard cell placement have been reported as well [17].

Simulated Evolution is a new heuristic for standard cell placement that combines the features of iterative improvement and constructive placement with the ability

to avoid getting stuck at local minima using a stochastic approach. The heuristic is based on an analogy between the natural selection process in biological environments and the method of solving engineering problems by iterative improvements. Descriptions of implementations of the evolution-based algorithm can be found in [18–21].

Recently, another placement algorithm called Genetic Placement has been proposed which uses the concepts of maintaining a set of solutions (population) and performing crossover and mutation operations on those populations to produce better populations [22, 23].

For more details on various sequential placement algorithms, the reader is referred to some excellent surveys in [1, 24–26].

1.4. Placement Cost Functions

Each of the above placement methods depends on the cost function employed in order to measure the acceptability of a current placement. Since the two-fold goal of cell placement is to minimize the placement area while ensuring the routability of the layout, cost functions have examined various criteria such as estimated wire length and cell congestion. One simple method for estimating the wire length is to measure the half-perimeter of a box which bounds the pins of a given net. Figure 2 graphically shows how the bounding box measure would be calculated. A more computationally intensive measure is to calculate the wire length of the minimal Steiner tree.

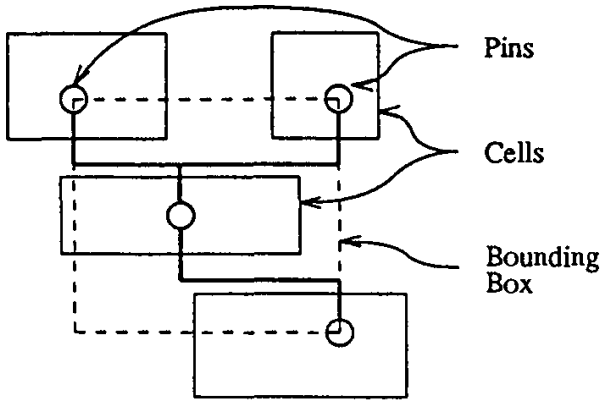


Fig. 2. Wire length estimation by bounding box.

One way to measure cell congestion is to calculate the number of nets that connect separate partitions of the set of cells. The goal is then to minimize the number of nets cut by a line separating the partitions. Figure 3 shows a high and low cost configuration for a small example circuit. Some placement algorithms try to minimize the total wirelength, or the congestion or some combination of both.

A simplified model of placement used by some sequential and parallel placement algorithms is as follows. Given a set of modules of equal size, and connection matrix

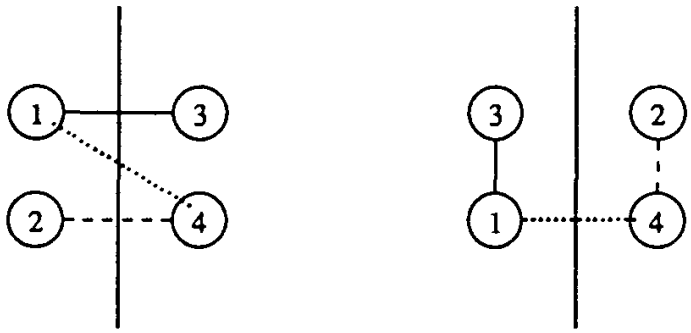


Fig. 3. Cell congestion estimation by net cut count.

describing their interconnections, the placement problem can be viewed simply as assigning the modules in rows and columns in an array of positions on a chip image. This assumes that multiterminal nets have been decomposed into a collection of two-terminal nets, hence a multiterminal netlist of connections can be represented as a set of two-terminal connections specified as a connection matrix $C = [c_{ij}]$ which gives the number of connections between module i and module j. Assuming fixed sized modules to be placed in fixed locations in the chip layout, a distance matrix $D = [d_{ij}]$ can be defined that measures the Euclidean distance between the modules i and j. The objective function that is minimized is $G = \sum_{i,j} c_{ij} * d_{ij}$.

1.5. Overview of Parallel Placement Algorithms

Having given a brief overview of the placement problem and various algorithms we will now describe some parallel algorithms that have been proposed for these algorithms. We will classify them into the following classes and describe the detailed algorithms for them in subsequent sections:

(1) Approach using pair-wise interchange.
(2) Approach using simulated annealing.
(3) Approach using simulated evolution.
(4) Approach using genetic algorithms.
(5) Approach using hierarchical decomposition.
(6) Approach using a combination of some of above techniques.

For each of the above classes of algorithms, we will discuss first the sequential algorithms on which they are based, and then describe the parallel algorithms. The parallel algorithms will be discussed for both shared memory multiprocessors, and distributed memory message-passing multiprocessors.

2. Parallel Placement Algorithms Using Pair-Wise Interchange

The first reported works in parallel placement algorithms were designed to run on two-dimensional arrays of processors by two groups of researchers, Chyan and Breuer [27], and Ueda, Komatsubara and Hosaka [28]. The parallel placement

algorithms by both groups of researchers used iterative improvement techniques using adjacent pair-wise exchange of cells. Even though the parallel algorithms were targeted to run on special purpose array processors, we will adapt and discuss a suitably modified algorithm for a general purpose parallel processor.

2.1. Overview of Sequential Algorithm

The adjacent pair-wise exchange method of placement improvement uses the simplified model of placing modules of fixed size whose connections are specified by a connection matrix representing connections using two terminal nets. The objective function to be minimized is the total wirelength using the sum of the Euclidean distances between modules weighted by the number of connections between modules.

The iterative algorithm proceeds as follows. First select a single pair of neighboring modules in a given sequential order and calculate the sum of wirelength associated with each module in the pair. If the total wirelength associated with the pair can be decreased by the exchange, these two modules are actually exchanged. Otherwise they are not exchanged. The above steps are repeated until no more improvement is obtained. This procedure is illustrated in Fig. 4 for a 4×4 array of modules. The module pairs that are chosen for exchange considerations are only immediate neighbors. Figure 5 shows an example ordering of node pairings for cases N even and odd, which will be executed in sequence from 1 to 24 for $N = 4$, and 1 to 40 for $N = 5$. Such an ordering guarantees that after a set of iterations involving repeated executions of pair wise swaps of categories (a), (b), (c) and (d), any module can move to any other location in the chip layout using a set of adjacent moves.

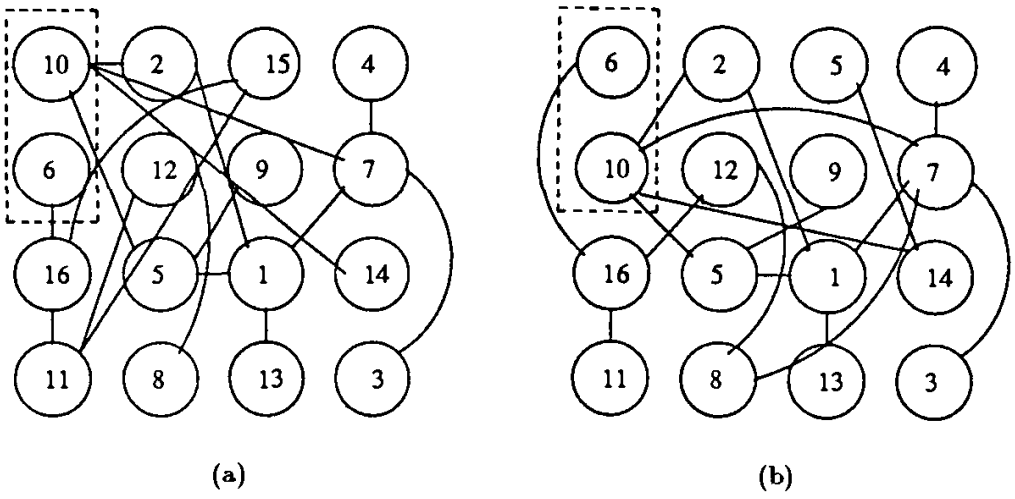


Fig. 4. Example of placement improvement by iterative improvement (a) Before exchange (b) After exchange.

2.2. Distributed Memory Parallel Algorithm

The parallel algorithm for the above adjacent pairwise exchange placement algorithm proceeds by distributing regions of the layout and the corresponding modules to different processors. For an $N \times N$ array of modules to be placed using P processors, the processors are arranged as an $\sqrt{P} \times \sqrt{P}$ array of processors. Each processor gets assigned a square subregion of chip corresponding to the subarray $\frac{N}{\sqrt{P}} \times \frac{N}{\sqrt{P}}$ of modules. The modules are initially placed through random assignment or some simple clustering algorithm. The parallel iterative improvement algorithm involves forming independent pairs of modules to be exchanged in parallel among the various processors. All pairs of module exchanges that are shown in ellipses in Fig. 5 for categories (a), (b), (c) and (d) can be performed in parallel. This is because the processors are accessing modules that are guaranteed to be independent by the spatial data distribution. During one such iteration, individual processors or pairs of processors make local decisions regarding whether or not to actually swap the pair of corresponding modules. After on such iteration, the state of the module placement needs to be informed to all the processors, hence a global update of all module locations is performed through broadcasting. This enables the processors in a subsequent iteration to properly calculate the wirelength costs of all the modules.

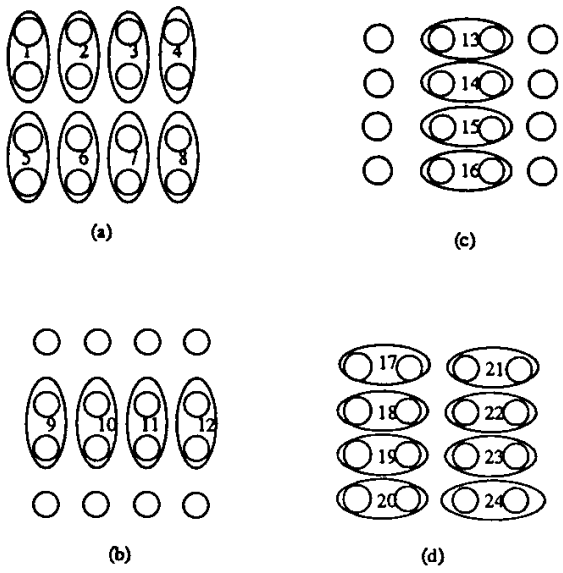
The procedures performed within each pair of processors during a single iteration of parallel processing cycle is as follows.

ALGORITHM PARALLEL-ITERATIVE-EXCHANGE;
Initially allocate layout regions and modules to each processor with an initial random placement. For I Iterations repeat

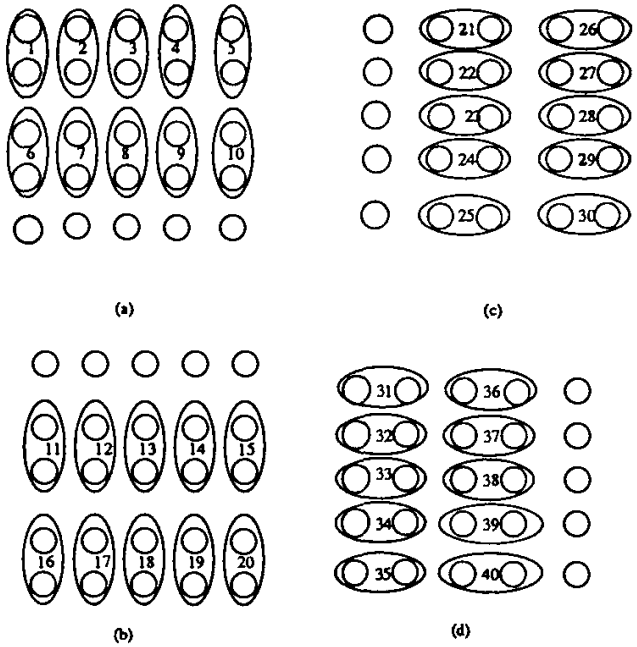
For Cycles (a), (b), (c) , (d) in sequence do

1. For category (a) .. (d), identify the pair of modules to form independent pair to be exchanged. If both the modules are within the same processor, calculate the amount of addition or reduction in wirelength associated with the module exchange locally.
2. If the pair of modules exist on different processors, identify neighboring processor to be the mates for category (a) .. (d) to form independent pair, and select modules to be considered for exchange. Identify one processor to be Master, other to be slave for each pair.
3. Each processor calculates the amount of addition or reduction of wirelength associated with the module when the module is exchanged with the module of present mate processor. The master processor receives the resultant data which is calculated and added to the value of the slave processor to determine whether to exchange the module locations. If there is a reduction in the wirelength an actual module exchange is marked.
4. For each processor, broadcast the information about the new module locations to the affected processors that are related to the exchanged modules.

Endfor I; Endfor cycles;
END-ALGORITHM



(a) Case where N is even.



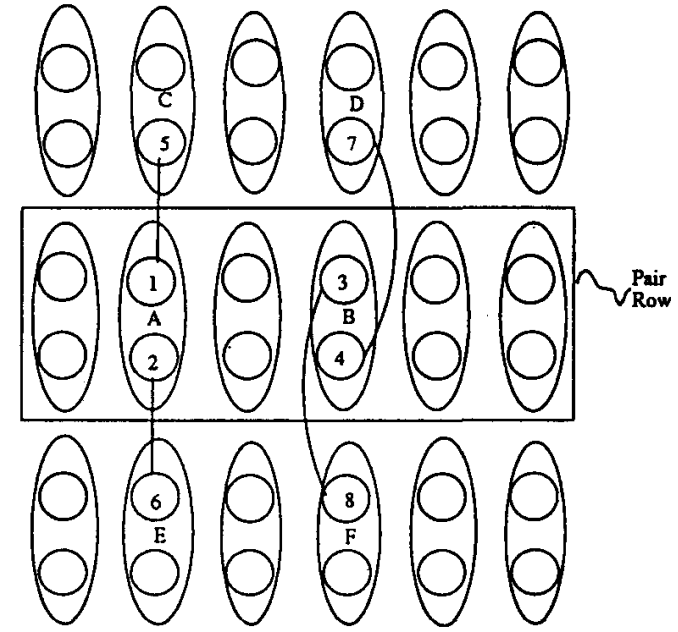
(b) Case where N is odd.

Fig. 5. Pair combination categories in adjacent pair exchange algorithm for N even and N odd.

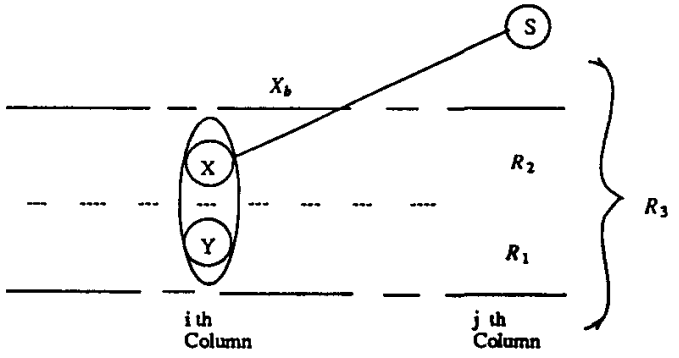
In the previous discussion it was assumed that all the selected pairs of modules are totally independent from each other in the parallel placement improvement process. By investigating the parallel improvement process in more detail Ueda *et al.* have identified [28] the cases where some module pairs become dependent upon the other pairs under certain conditions. There are two cases to consider.

(1) connections between modules in different pairs of rows (2) connections between modules in same pairs of rows.

Case 1. Connections between different pairs of rows. Let us consider the examples of Fig. 6(a). In this case pair A judges that the sum of its wirelengths will increase by "2" if module 1 and module 2 are exchanged. Therefore the modules are not actually exchanged. This judgement is obviously correct independent of whether or not their related modules 5 and 6 in pairs C and E are exchanged.



(a)



(b)

Fig. 6. (a) Independent pair example (b) Case where connections exist between different pairs of rows.

For the general case shown in Fig. 6(b), let $R1$ and $R2$ be the regions with pair (XY) and $R3$ be the remaining region. Let S be an arbitrary node in region $R3$. If modules X and Y are to be exchanged, then the sum of wirelengths from modules X and Y will decrease after the exchange whether or not the module S is moved within $R3$, as long as it does not cross $R1$ and $R2$, which cannot occur by the nature of the restricted parallel exchange patterns. Therefore the exchange judgements are always correct for these cases.

Case 2. Connections in the Same Pair Row. When a given pair of nodes to be exchanged has connectivity with nodes within the same pair row, errors in judgement may occur. Let us consider a general case of Fig. 7, which shows various cases of node pairs (XY) being considered for exchange while having connections to nodes in the same regions $R1$ or $R2$. In Fig. 7(a), the node X has a connection to node T in region $R2$. When node pair XY is exchanged, if node T is not exchanged as well, the expected amount of wire length reduction is obtained. If node T is exchanged in addition, an extra amount of reduction is obtained. Hence this case does not produce any incorrect judgement. Next consider case Fig. 7(b) where node X has a connection to node U in region $R1$. In this case under exchange of XY, a wire length reduction of 1 is expected. If node U moves as well, the reduction becomes zero. If there are two or more connections to nodes in region $R1$ as shown in Fig. 7(c), then there is an incorrect judgement. Similarly, Fig. 7(d) has a possibility of incorrect judgement.

Neither groups of researchers, Ueda *et al.* [28], nor Chyan and Breuer [27], have reported on actual implementations of the parallel algorithms on real parallel machines. The authors have reported simulation results on some randomly generated circuits of on a 4 × 4 array, 6 × 6 array, etc. The data obtained by the simulation programs showed that through parallel processing they have been able to achieve final placement quality as good or a little better than the corresponding sequential algorithms. We now evaluate the parallelism theoretically exploited by this algorithm. With the sequential algorithm, only a single pair of modules is selected and processed at a time, while with the parallel processing algorithm executing on P processors, on the average, $\sqrt{P}(\sqrt{P-1})/2$ pairs of modules are processed concurrently. Since numerous modules are processed concurrently, it leads to the reduction of the number of iteration cycles consumed to obtain the final placement compared to the sequential algorithm. The number of iteration cycles which are consumed for obtaining a final placement with the sequential algorithm increases proportional to N^2, whereas the number of iteration cycles for the parallel algorithm increases as P. From the experimental results, the theoretical cycle reduction rate of $\sqrt{P}(\sqrt{P-1})/2$ has been verified. However, this reduction rate value does not necessarily correspond to processing time reduction because the period of the parallel processing cycle is generally different from the sequential processing cycle. This is because the module location update phase in a parallel processor can be an expensive synchronization overhead resulting in less than expected speedups.

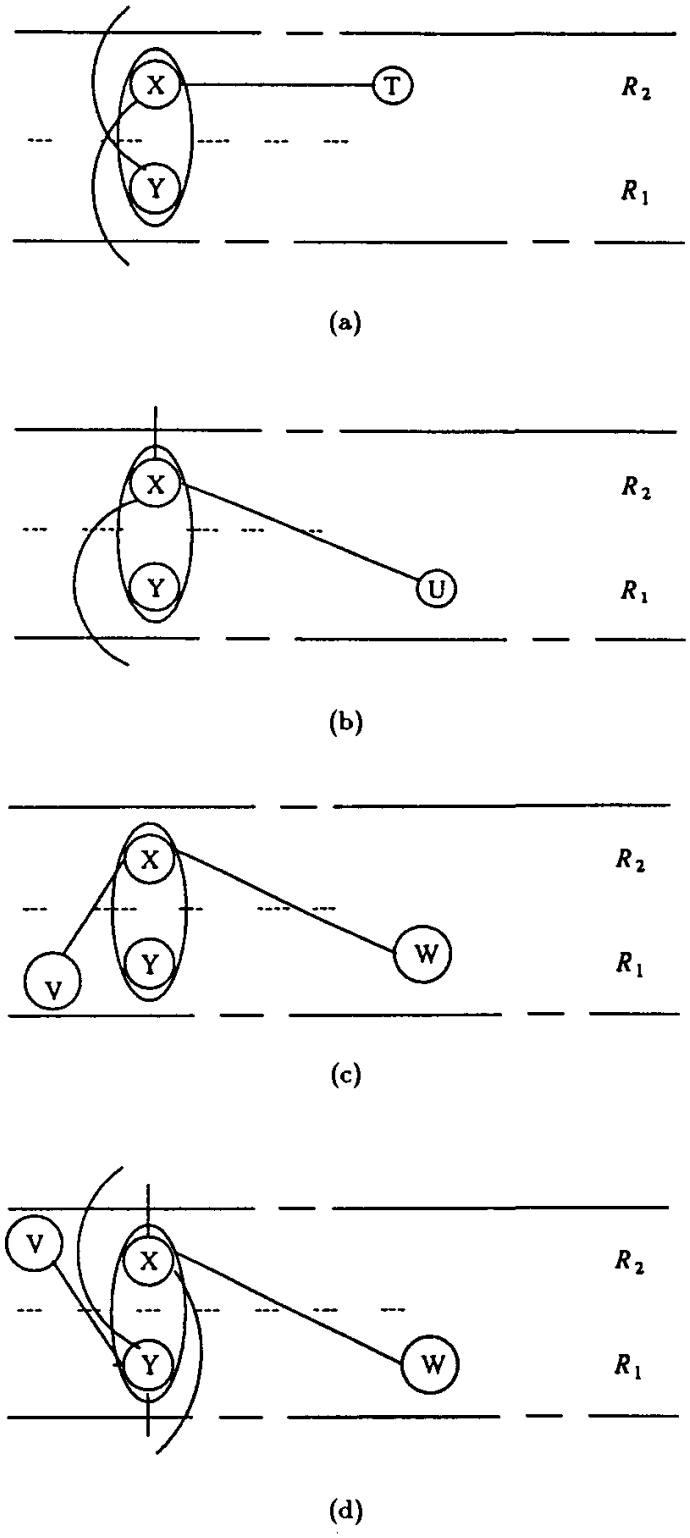


Fig. 7. Case where connections exist at same pair row.

3. Parallel Placement Algorithms Using Simulated Annealing

Simulated annealing is a powerful technique that has been used to solve combinatorial optimization problems using a probabilistic hill-climbing algorithm with the added ability to escape from local minima in the search space [12].

A general simulated annealing algorithm proceeds as follows:

```
ALGORITHM SIMULATED ANNEALING;
    Start with an initial solution (state) S in the search space.
    Set T = T0 (initial temperature).
    Repeat { /* outer loop */
            Repeat while (not equilibrium) { /* inner loop */
            Generate a move to perturb S to generate a new state, Sn.
                    Δ E = E(Sn) − E(S).
                    If (Δ E < 0 )
                      Accept move, i.e., replace S with Sn.
                      Else with probability e^(−ΔE/T),
                        Accept move, i.e., replace S with Sn.
                      }
                      Update T (control temperature)
                      End Repeat /* inner loop */
    } Until "frozen" (termination condition) /* end outer loop */
END ALGORITHM;
```

In any simulated annealing algorithm, four important criteria are the choice of the initial temperature, the equilibrium detection condition at a particular temperature, the rate of decrease of the temperature, and the frozen or termination condition. The set of these criteria is referred to as the *annealing schedule*. Most implementations of simulated annealing use a fixed sequence of temperatures derived empirically [12, 15]. Huang *et al.* have proposed an adaptive cooling schedule based on the characteristics of the cost distribution and the annealing curve itself (average cost vs $\log_1 0$ temperature) [13].

Theoretical studies have shown that a global optimum can be reached with unit probability provided a set of conditions on the annealing schedule are satisfied which might take infinite time. In practice, since the computations have be carried out in finite time, simulated annealing converges to near-optimal solutions. The real disadvantage of the simulated annealing approach is the massive computing time required to converge to a near-optimal solution. Various approaches have therefore been proposed to speed up the simulated annealing process in the sequential algorithm such as changing the annealing schedule and investigating better move sets.

3.1. Overview of Sequential Algorithm

The simulated annealing technique has been successfully applied to the standard cell placement problem in a program called TimberWolf [15, 29]. The cost function used in the original version of the Timberwolf algorithm (version 3.2) for standard cell placement [15] consists of three parts:

1. Estimate of the wire length of all nets as the half perimeter of the bounding box which is the smallest aligned rectangle which contains all terminals in a net.
2. Overshoot or undershoot of each row length over the desired row length;
3. Area overlap between cells in the same row.

Two types of moves are used to generate new configurations. Either a cell is chosen randomly and displaced to a random location on the chip, or two cells are selected randomly and exchanged. The ratios of displacement to exchange moves is set to about 5:1. A temperature dependent range limiter is used to limit the distance over which a cell can move. Initially, the span of the range limiter is set such that a cell can move anywhere on the chip. Subsequently, the span decreases logarithmically with temperature. The annealing schedule is a fixed one. At each temperature, a fixed number of moves per cell is attempted. The initial temperature is set to a very high fixed temperature, and the final temperature is a very low fixed temperature. The cooling schedule is represented by $T_{i+1} = \alpha(T) \cdot T_i$, where α varies between 0.8 to 0.95.

Other implementations of simulated annealing algorithms for cell placement have been reported. For example, in a future version of Timberwolf (version 4.2), several improvements have been made to improve the runtime performance. The cost function is the same as the earlier version. However, the move generation and cost computation has been changed. Each row is divided into nonoverlapping bins, using which the computation of the row overlap and row overshoot functions become very efficient. The following procedure is adopted for move generation. A cell is selected randomly, and a random location is selected as a destination. If the destination is vacant, a displacement is performed, otherwise an exchange is performed. The temperature profile is again a set of fixed temperatures but over a smaller range.

In the next section we will investigate techniques for speeding up simulated annealing algorithms for cell placement by running them on parallel processor systems. For the purpose of this discussion, we will use the Timberwolf (version 3.2) approach discussed above. The ideas of the more recent versions of sequential simulated annealing algorithms can be easily incorporated into the parallel algorithms as well.

3.2. Overview of Parallel Algorithms

There are various ways to apply parallelism in simulated annealing.

A. *Use parallel evaluation within each move.* In this approach, each individual move is evaluated faster by breaking up the task of evaluating a move into subtasks and allocating various subtasks to different processors.

B. *Use parallel evaluation of multiple moves.* This approach is general for any application of annealing. Unfortunately, there is a problem with using parallelism to propose and evaluate several moves simultaneously since global information about the state is required in order to evaluate the cost function. The cost function calculations may be incorrect due to interacting moves, similar to the cases discussed earlier for the parallel algorithm using pair-wise exchange [28]. If two processors are simultaneously considering moves involving different cells, one processor may move a cell after the second processor reads the state of the layout. The second processor now has inaccurate information about the positions of the cells and will use this out-of-date information to compute the cost of the proposed move. The resulting evaluation will be wrong and may lead to wrong decision in accepting or rejecting a move. There are two sub-approaches for handling move interactions in parallel moves.

B.1 *Use parallel evaluation of multiple moves and acceptance of moves that do not interact.* In this approach, a set of moves can be identified either statically or dynamically such that they do not interact and apply parallelism to those moves. The convergence characteristics of the parallel algorithm is identical to the serial algorithm. However, since many possibly good moves are left unused, a majority of computations are wasted. Hence the effective useful parallelism is limited.

B.2 *Use parallel evaluation and acceptance of multiple interacting moves.* In this case multiple moves are evaluated for acceptance based on inaccurate information, namely some errors in computations of cost functions are allowed. Here, there can be problems with move interactions, affecting the convergence characteristics of the parallel algorithm. However, this approach has the scope for maximum parallelism. In such schemes, a large number of parallel moves are evaluated and accepted on the basis of the past state information. The new cell positions are updated after every set of parallel moves.

Figure 8 illustrates the various ways one can apply parallel processing to speed up simulated annealing as discussed in Kravitz and Rutenbar [30]. We will now look at each of the proposed algorithms in a little more detail. An excellent review of parallel simulated annealing algorithms for placement appears in [31].

3.3. Approach Using Parallel Evaluation Within Each Move

In this approach, parallelism is used to generate a move faster. The work of a move consists of selecting a feasible move, evaluating the cost changes, deciding to accept or reject, and perhaps updating the global database. Kravitz and Rutenbar [30] have proposed two different approaches to move decomposition. In one approach, called object decomposition, the responsibility of a set of objects is

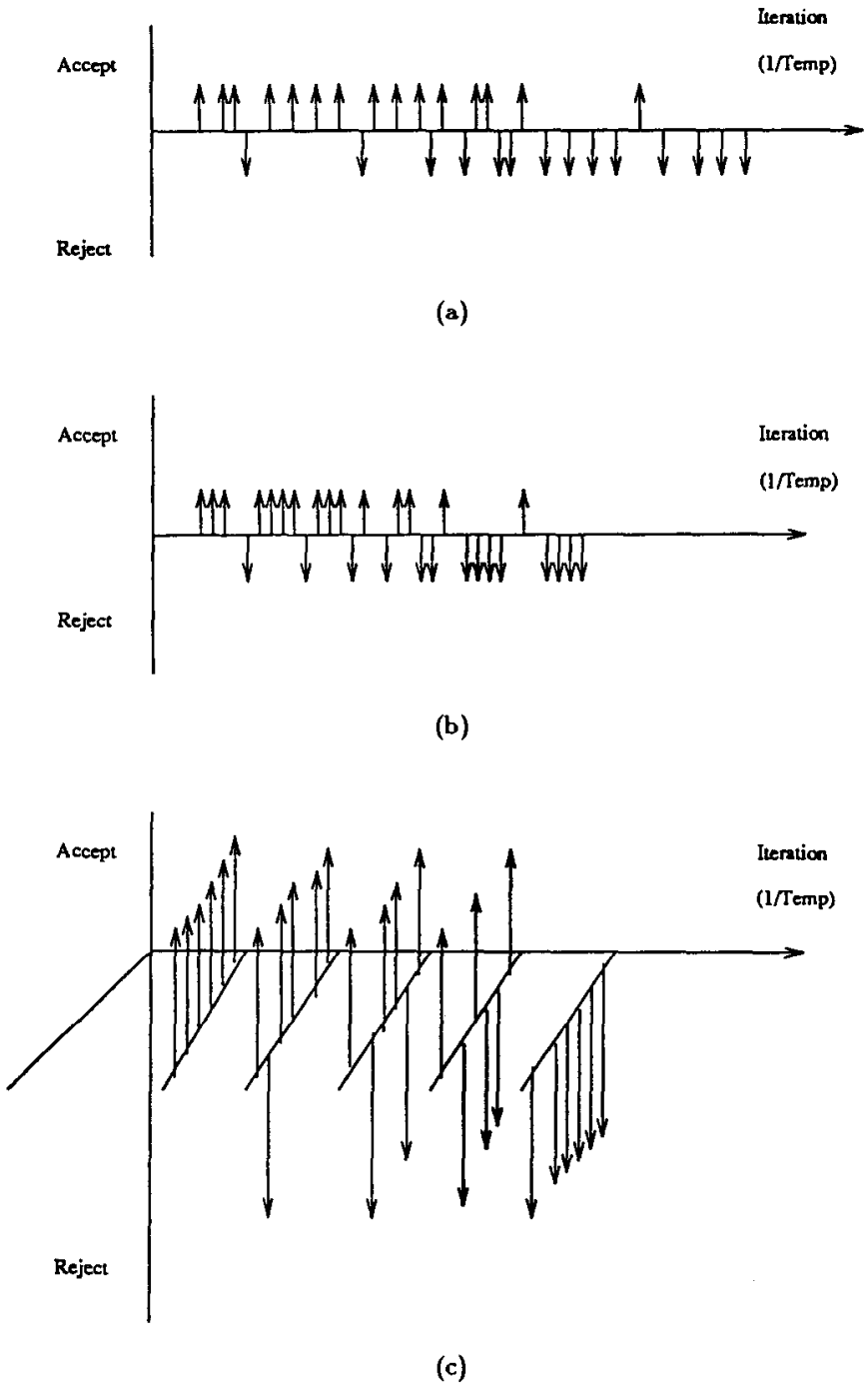


Fig. 8. Approaches to parallel annealing. (a) Serial annealing (b) Accelerating each move (c) Parallel move evaluation.

delegated to a particular processor. For example, one can divide the cells and nets into groups and assign each group to a processor. All operations with respect to moving, accepting, evaluating and updating that set of cells is done by that processor. Another approach, called functional decomposition, delegates the individual portions of work such as wirelength evaluation and updating to different processors.

The authors have proposed further divisions within the object and functional decomposition into static and dynamic cases. If the distribution of objects to processors occurs at initialization and never changes, the strategy is static. If the objects can be reassigned during execution, the scheme is dynamic. Similarly, within functional decomposition, one can have static and dynamic cases. The first method uses dynamic scheduling and many small subtasks to evaluate the cost function and update the database. The second method which used static scheduling and fewer subtasks dedicated to specific processors. Such approaches to parallelization are limited to shared memory algorithms only, and have limited scope for parallelism.

3.3.1. Shared memory parallel algorithms

Kravitz and Rutenbar have implemented their algorithms on a 4 processor VAX cluster [30]. Below we describe some of the implementation details. In the parallel implementation, a master process forks several slave processes equal to the number of physical processors (=4 in their implementation), and itself performs all the serial parts of the computation, while the parallel slave processes perform the computation-intensive annealing tasks.

The dynamic functional move decomposition partitions each move into subtasks, and distributes the jobs to processors. Figure 9 shows the details of the dynamic

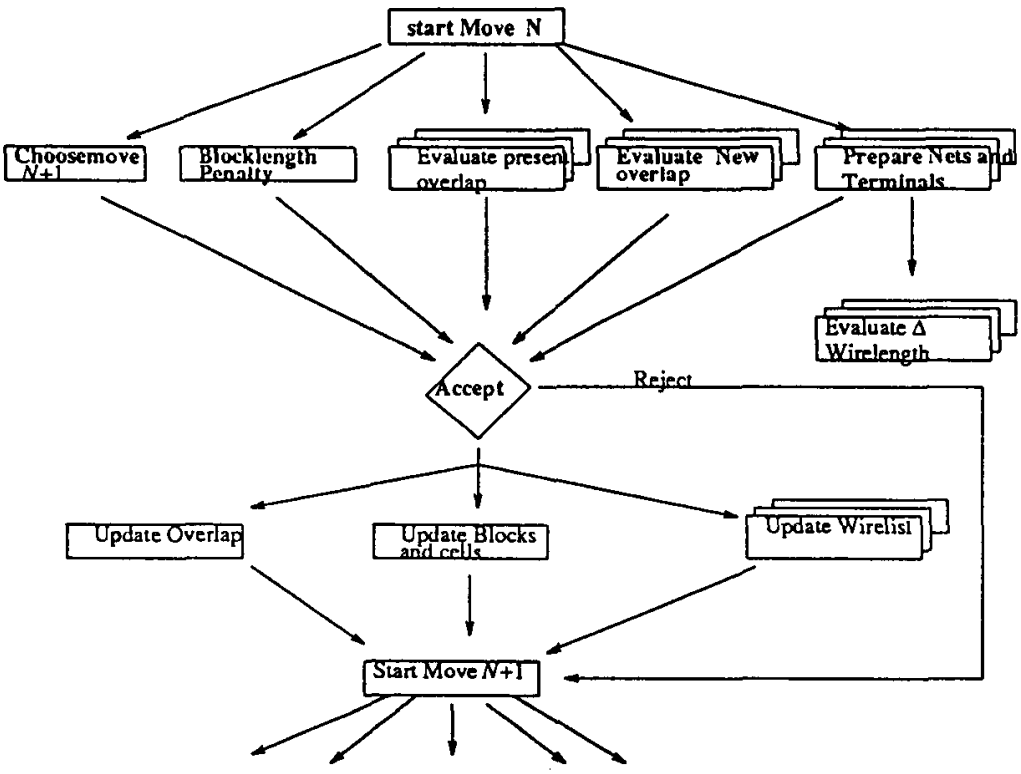


Fig. 9. Dynamic Functional Move Decomposition.

functional move evaluation. Each move is dynamically divided into several jobs that are inserted into a shared task queue data structure. Real executable tasks from the operating system acquire these structures and execute the necessary work. The static functional decomposition implementation eliminates the need for synchronization of the subtasks of the dynamic scheme by statically dividing the work into three parts. Figure 10 shows the division of labor in the Static function implementation. From the experimental results it was determined that the dynamic functional decomposition was not successful because scheduling the many subtasks added too much overhead and created a bottleneck at the task queue. The static functional decomposition used fewer subtasks, was more successful, and gave a speedup of two on 3 processors.

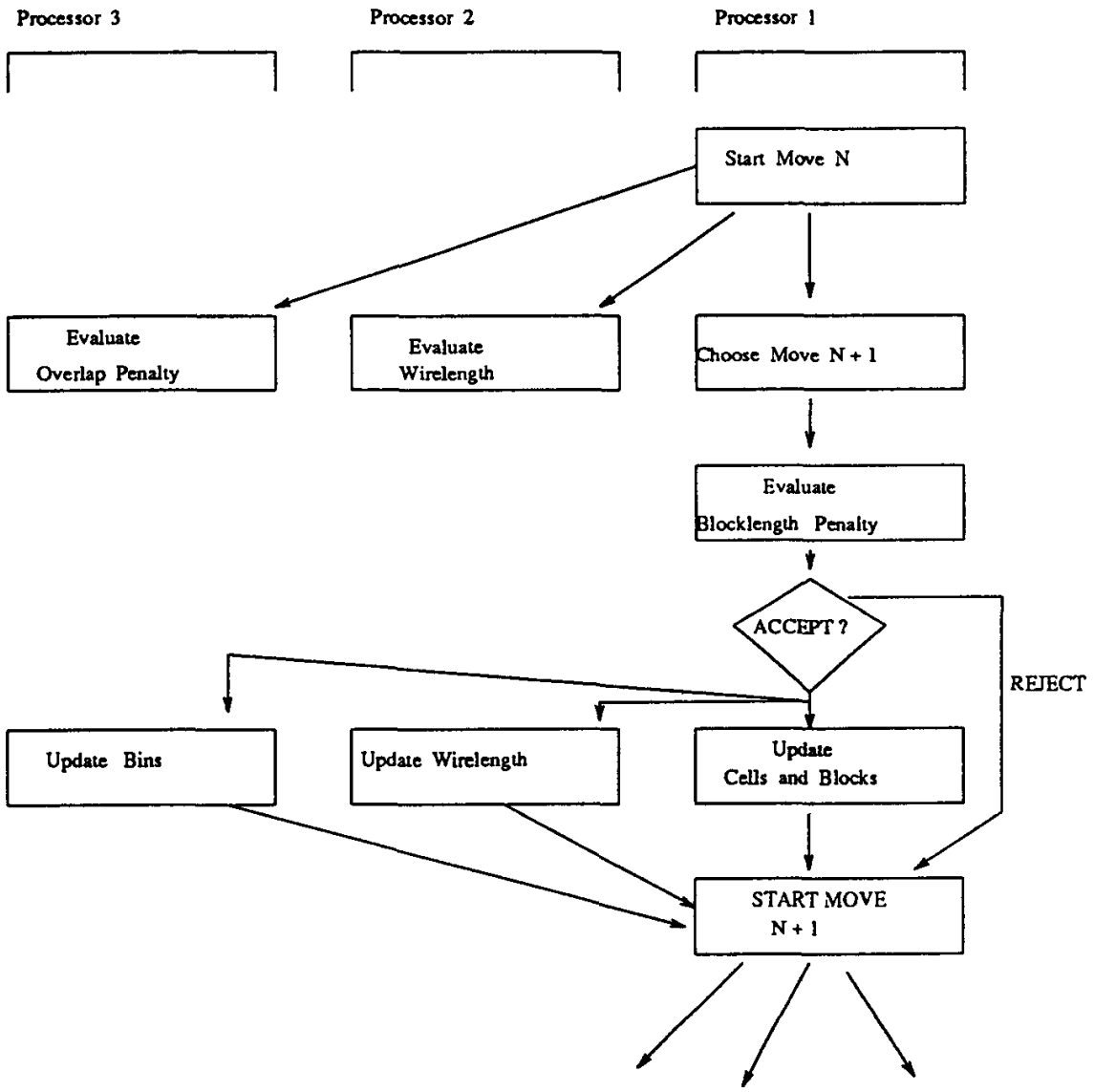


Fig. 10. Static function implementation.

One should note that such approaches to parallelization are limited to shared memory multiprocessors only, and have limited scope for parallelism (speedups of about 3–4).

3.4. Approach Using Parallel Evaluation and Acceptance of Noninteracting Moves

In this approach, one processor is applied to each move so that moves can be generated and evaluated simultaneously. There are however several alternatives that one has to choose from. First, each processor could choose moves independently from the entire set of available moves. Secondly, one could bias each processor's move choices using a spatial decomposition where each processor is assigned to perform moves within a region. We will briefly review some of the parallel algorithms that have been reported on shared memory and distributed memory multiprocessors to address these issues.

3.4.1. Shared memory parallel algorithms

Rutenbar and Kravitz [30] proposed the use of parallel evaluation of moves, but accepting only a *serializable set of moves.* Suppose a set S of moves is evaluated in parallel. Let a subset S^{ni} be a subset of noninteracting moves in S. By concurrently applying all the moves to the same initial configuration, one reaches at the same final configuration by applying the moves in any order. Hence the moves are serializable. To gain the maximum parallelism, one has to determine a maximum set of serializable moves. However, the authors noted that the determination of the largest set of serializable moves is itself a hard problem. Hence they resorted to the heuristic that only one of the "acceptable" moves is really accepted. This approach has the penalty of sacrificing some potential parallelism.

In the Parallel-Moves Strategy implemented by Kravitz and Rutenbar [30], the parallel moves are evaluated asynchronously. All the moves need not start and finish at the same time. If all moves are rejected, parallel evaluation proceeds with no further synchronization. There is a synchronization after the first acceptable move is found. A process can prevent other processes from starting the evaluation of a move by setting a lock. Once a process has begun evaluating a move, it can terminate in one of three ways: reject the move, accept a move, or abort a move.

Overall, their implementation for parallel annealing showed a speedup of about 2 to 2.5 times faster than their own version of simulated annealing based on TimberWolf3.2 that allowed only two types of moves, cell displacement and exchanges. They have not reported any results on the quality of results produced by their parallel algorithm.

Darema *et al.* have reported a different parallel algorithm for simulated annealing on a shared memory multiprocessor [32]. They considered a simplified problem of placing a number of equal sized modules connected by multiterminal nets on a chip represented as a rectangular array. The entire grid area is divided into vertical and horizontal channels around the module locations representing the wiring areas. The

cost function is $E = L + wC$, where L is the total wirelength as measured by the half perimeter of the bounding box of nets, and C is the congestion, as measured by counting the number of nets above a certain threshold that crosses each row or column. The evaluation of the congestion was done with the help of a histogram which stores the wire count for every row and column.

In the parallel implementation, each processor has a local copy of the histograms. A global copy is kept in shared memory as well. Since multiple processors attempt to perturb the configuration, it is important to ensure that no more than one processor attempts to move at any instant of time. A lock is therefore associated with each cell. A processor attempts a move by selecting two cells at random. The pair of cells and also the cells connected to those cells are locked to ensure that another processor will not attempt to move them simultaneously. The algorithm uses the local copy of the histograms to compute the change in energy. This method guarantees that the global histogram will be correct once all the processors have finished. It also guarantees that the wirelength calculations will be correct. Before a processor proposes a move, it first checks to see if the lock associated with the cell or other cells connected to the nets belonging to the cells are free. If so, it sets the locks in those cells, and proceeds further. If the lock is not free, then the processor tries at random to find another cell whose corresponding locks are free. The effect of a move proposed by a processor is evaluated by it under the assumption that no other perturbations of the configuration are being accepted at the same time by any other processor. If the move is accepted, the processor updates the global copy of the histogram by adding or subtracting one from the correct rows and columns using the *fetch-and-add* operation. Fetch-and-add is an atomic operation that merges operations bound for the same location and returns the result in the same time that it takes to execute one operation. Since a hardware implementation of the fetch-and-add operation in any multiprocessor did not exist at the time, the authors have done evaluation of their parallel algorithm using a simulated environment. The cycle of propose-in-parallel, evaluate-in-parallel, accept-in-parallel is repeated until the inner loop criterion is satisfied.

The asynchronous moves create a problem in that information in the local histograms can be both out of date and inconsistent even though the local copies are updated with the global copies. Even though the error in the computation in the wirelength is avoided by locking affected cells, errors in congestion estimates cannot be avoided. The congestion value might be wrong because moving cells in unconnected nets can change the number of wires that cross a certain row or column. When the histograms are updated after a cell is moved, the algorithm assumes that the cell on the other end of the net is stationary. If in fact, it is not, then the histogram updates are wrong. These errors are cumulative and the algorithm cannot correct the errors by waiting for all the processors to complete computation. The authors developed a variant of this method to control the accumulation of error. In this method, the algorithm stops all processors at the end of each iteration and computes new histograms from the actual locations of the cells.

The authors experimented with both the runtimes and the quality of the results of the parallel algorithm for a small 81 cell example on a 9 × 9 grid for which the optimal solution was known. They investigated the variations of the average energy versus the temperature. The parallel algorithm showed similar results to the serial algorithm because it was trying to follow the uniprocessor algorithm. On a simulated 16 processor multiprocessor, the method gave a speedup of about 7. This was assuming that the machine has the support of the Fetch-and-add instruction. As most existing shared memory machines do not have such an instruction, the overheads would be more (and the speedup less) on a real machine.

3.4.2. Distributed memory parallel algorithms

Banerjee and Sargent have described [33] a parallel algorithm for performing the standard cell placement suitable for execution on a hypercube based distributed memory multiprocessor using a variation of the TimberWolf algorithm. Given the problem of placing a set of standard cells on a hypercube of 2^d processors, the basic idea used in this algorithm is to assign a row or set of rows of the VLSI chip image including all cells in that row to each processor in the hypercube, and to have parallel movements of cells between rows. The rows are mapped to the processors of the hypercube using a binary reflected gray code mapping [34]. For example, for a standard circuit with 8 rows of cells, each processor in a 8-processor hypercube will be assigned an entire row as shown in Fig. 11.

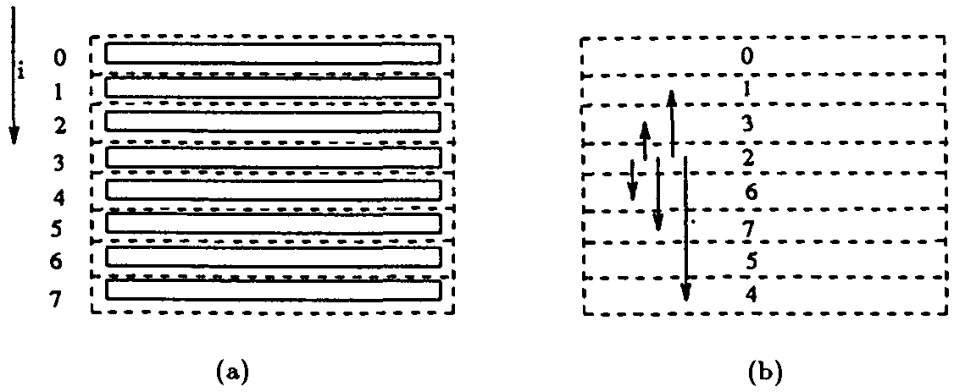


Fig. 11. Row partitioning of an 8 row circuit to 8 processor hypercube. (a) Row partitioning. (b) Region assignment to processors.

Cells are assigned an initial position, but are free to migrate between processors as they move around the chip area. Using a distributed data structure, this algorithm is able to support parallel moves. Processors pair up to evaluate a single move type — either a displacement or an exchange of cells. One can therefore guarantee that the cells that will be selected for parallel moves by different processors are different. For a 2^d processor hypercube, upto 2^{d-1} parallel moves (cell exchanges or displacements) are evaluated and accepted/rejected. After one set of parallel moves, all the new cell locations are broadcast to all the processors.

The advantage of the row partitioning scheme is that each processor has all the relevant neighborhood cell placement information and has therefore exact knowledge of the two penalty cost components (the row overlap and the cell overlap). Hence, under parallel move evaluation and acceptance, the only error that can accumulate due to parallel moves is error in total wire-length. The authors proposed an efficient circuit pre-processing algorithm called **Heuristic Cell-Coloring**, that completely eliminates the error in the wire-length computation as well. This is achieved by identifying sets of noninteracting cells, i.e., cells that do not have any common nets. Noninteracting cells can be moved repeatedly, and in parallel, without any accumulation of cell position misinformation in the distributed database.

Finding sets of unconnected nodes in an arbitrary graph is analogous to graph coloring. By viewing the circuit description of a standard-cell circuit as a graph where cells correspond to nodes and nets correspond to edges, the graph is colored so that no two connected vertices are the same color. All vertices (cells) of the same color are non-interacting, and can be moved repeatedly between updates without any error accumulation. Though optimal graph-coloring for arbitrary graphs is NP-complete, fast heuristic graph-coloring methods are available for graphs that are not "pathological cases". The authors have reported on parallel annealing algorithms using three procedures for graph coloring.

The authors reported on the results of an implementation of the above algorithm on a 16 processor Intel iPSC/2 hypercube. Speedups of about 12 on 16 processors were reported. However, the qualities of the results was actually inferior to the serial simulated annealing algorithm. This can be attributed to the following reasons. While in the serial algorithm, cells can be selected randomly and can be moved to random locations in the chip, the above algorithm constrained the randomization in the search process in two ways. First, only cells of a particular color were chosen for displacement or exchange at a time. Secondly, the cells were only allowed to move from the geographically assigned region of one processor to the region of another processor with which the processor was communicating at the time. These two restrictions actually affected the full effectiveness of the simulated annealing search procedure. Of the two reasons, we believe that the cell coloring is more restrictive and damaging to the convergence process, since other spatial decomposition techniques will be discussed later, and will be observed not to affect the convergence of the solution significantly.

3.5. Approach Using Parallel Move Evaluation and Acceptance of Interacting Moves

It was noted earlier that independently evaluated parallel moves may interact, and hence give erroneous accept/reject decisions. While the previous approaches to parallel move evaluation in annealing either restricted the number of accepted moves to unity, or accepted multiple moves if they were non interacting, a variety of schemes have been proposed for allowing multiple interacting moves to be accepted

but somehow try to bound the errors in the move evaluation. We will discuss several such approaches below, both for shared memory multiprocessors and for distributed memory multiprocessors.

3.5.1. Shared memory parallel algorithms

Darema *et al.* [32] have proposed a parallel algorithm for cell placement of equal sized cells and multiterminal nets with parallel move evaluation and acceptance on a shared memory multiprocessor as follows. The algorithm is identical to the one discussed earlier for noninteracting move acceptances, except that instead of locking all the cells connected on the nets associated with the pair of cells being considered for a move, only the pair of cells considered for the move is locked. The authors have performed some experiments on a sample 81 cell placement example on a simulated 16 processor multiprocessor. The results of the layout quality were always inferior to that produced by a sequential algorithm due to the acceptance of interacting moves but the speedups measured were much higher; speedups of about 11 on 16 processors were obtained compared to the speedups of about 7 using the full locking schemes.

Natarajan and Kirkpatrick have reported a different parallel simulated annealing algorithm on a shared memory multiprocessor using a decomposition approach in [35]. The layout area is divided into many subareas, and placement within each subarea is improved by a separate processor. The partitioning into subareas is varied during the run of the algorithm to ensure unrestricted movement of the circuit modules across the entire chip area. Some details of the decomposition approach are provided below.

```
ALGORITHM PARALLEL-DECOMPOSE;
Serial Begin;
      Start with an initial solution (state) S in the search space.
      Partition the set of modules into P domains A1,.., AP.
      Set T = T0 (initial temperature).
Serial End;
Repeat{ /* outer loop */
      Assign Domain Ai to processor Pi;
      Repeat-IN-PARALLEL while (not equilibrium) { /* inner loop */
            /* each processor works on own domain */
            Propose-in-parallel a MOVE to generate a new state, Sn
            in own domain Ai.
            Evaluate ΔE = E(Sn) - E(S).
            If (ΔE < 0 )
              ACCEPT-IN_PARALLEL Sn.
            Else ACCEPT-IN_PARALLEL move with probability e^(-ΔE/T),
```

```
    }
    Serial Begin;
        Update T and GLOBAL STATE;
        Modify domain partitioning;
    Serial End;
    End Repeat /* inner loop */
} Until "frozen" (termination condition) /* end outer loop */
```

Computation in the inner loop is shared among multiple processors as follows. Each processor proposes a move in parallel with other processors. Processor P_i is allowed to choose modules within its own domain A_i. The effect of a move proposed by a processor is evaluated under the assumption that no other perturbations of the configuration are being accepted simultaneously by another processor. If a move is accepted by a processor, it accepts an ACCEPT-IN-PARALLEL step. This step consists of modifying the globally shared values of the histogram values of channels, and the cost function.

The choice of a partitioning strategy is discussed next when a parallel algorithm is run with K processors. For each K, this approach uses a predetermined set$\{D_1, D_2, \ldots, D_{m_k}\}$, consisting of m_k different K-way partitions of the chip. These are illustrated in Fig. 12 for four different 3-way decompositions of the chip area. For each iteration of the inner loop, the processors use a fixed partition. The cycle of partitions is repeated so that a cell can go potentially from any location to any other location of the chip. The set of partitions is not unique. Some of the possible choices include dividing the chip into rectilinear regions that are modified

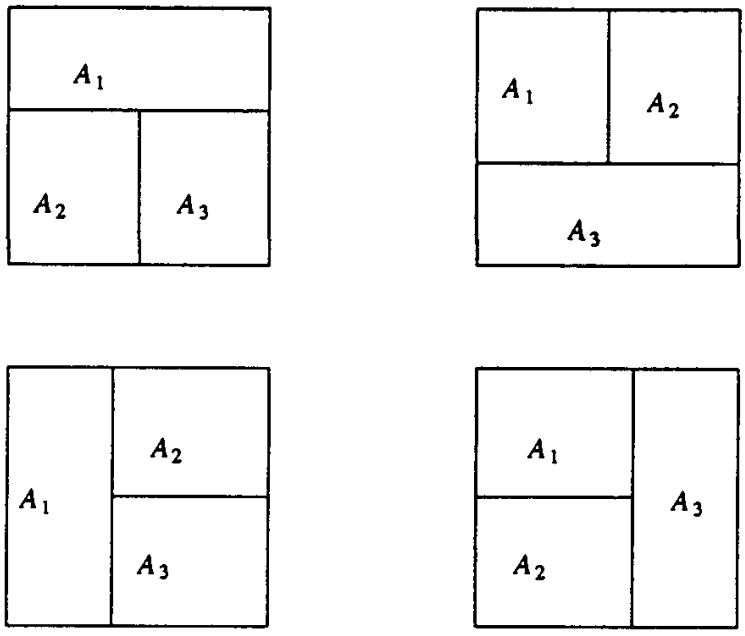


Fig. 12. Four different 3-way decompositions of chip area.

at runtime to ensure unrestricted movement of cells. Some of the experiments run by the authors did not show any one way of partitioning to be consistently better than the rest. Therefore this is left to the user.

Casotto *et al.* have proposed another variation of parallel simulated annealing algorithm for the macrocell placement problem [36]. The macro-cell placement problem involves a set of rectangular modules of arbitrary sizes and aspect ratios that are to be placed on a chip area such that they do not overlap, and that there is enough space left over for routing the connections, and such that overall layout has minimum area of a bounding box. A serial simulated annealing algorithm for this problem is described below.

The cost function is chosen to be the weighted sum of the estimated wire length (as estimated by half-perimeter of bounding box of the terminals of each net), the total area of the chip (as measured by area of smallest rectangle containing all the cells), and the total overlapping area of the cells. This approach does not really perform a true floorplan in that the aspect ratios of the cells are fixed and not variable as in the floor plan example.

The basic approach in parallelizing this algorithm is to partition the set of cells among the processors in such a way as to minimize the interaction among parallel cell movements and hence reduce the error. The initial partitioning is random but it is successively updated according to a criterion to be described later.

The parallel algorithm is as follows.

```
ALGORITHM PARALLEL-MACRO CELL;
Initially partition cells to processors.
Repeat outer loop
        For All processors in parallel do
            Repeat inner loop
            Select cells to move within processor ownership
                to any physical location in chip area
            Evaluate in parallel change in cost function
            Accept/Reject move in Parallel
            End Repeat
        End Forall
END ALGORITHM;
```

In the implementation of this algorithm on a shared memory multiprocessor, a single copy of the current state of the chip is kept in global memory. The cells are distributed evenly among the processors. Each processor asynchronously attempts to move a cell assigned to it either by displacing the cell, by rotating the cell, by mirroring the cell, or by exchanging it with another cell. Displacements, rotations, and mirroring require not synchronization. If a processor attempts to exchange a cell with another cell belonging to a different processor, it stops the other processor before completing the move. Since the ratio of displacements to exchanges are 10:1 the synchronization overheads are relatively low.

The authors attempt to minimize the error in computing the cost function by assigning the cells to processors such they rarely interact. The cost functions consists of a total estimated wirelength, the overlap area among cells, and the total area of the chip as measured by the area of the enclosing rectangle. Errors in cost function occur because the total wirelength was not computed accurately, or because the cell overlap estimates were incorrect. The authors tried to limit the error in overlap computations and area computations by assigning cells that are physically adjacent in the chip area to the same processor. Since cell locations dynamically change during the placement algorithm, the authors proposed the use of another simulated annealing based algorithm for assigning the cells to the processors that performs the cell assignment to processors using this clustering concept. This algorithm runs concurrently with the placement algorithm and periodically is used to rearrange the cell ownerships among processors.

The authors reported results on various circuits on an 8 processor Sequent Balance 8000 multiprocessor. The quality of results produced by the parallel algorithm was similar to the quality of results produced by a serial algorithm. Also the reported speedups were about 6 on 8 processors.

3.5.2. Distributed memory parallel algorithms

Banerjee, Jones and Sargent have described [33, 37–39] parallel algorithms for performing the standard cell placement suitable for execution on a hypercube based distributed memory multiprocessor using a variation of the TimberWolf algorithm. Given the problem of placing a set of standard cells on a hypercube of 2^d processors, the basic idea used in this algorithm is to assign a sub-area of the VLSI chip image including all cells in that sub-area to each processor in the hypercube, and to have parallel movements of cells between sub-areas. The subareas are mapped using either grid-wise or row-wise partitioning scheme. Cells are assigned an initial position, but are free to migrate between processors as they move around the chip area. Using a distributed data structure, this algorithm is able to support parallel moves. Processors pair up to evaluate a single move type — either a displacement or an exchange of cells. Only processors that are connected by direct hypercube links are paired up to evaluate a parallel move in the grid-wise partitioned algorithm. One can therefore guarantee that the cells that will be selected for parallel moves by different processors are different. For a 2^d processor hypercube, upto 2^{d-1} parallel moves (cell exchanges or displacements) to be evaluated and accepted/rejected by assuming that there is no interaction among the moves. After one set of parallel moves, all the new cell locations are broadcast to all the processors. In practice, there will be some interaction between the individual moves, hence there are some errors in the evaluation of the cost functions. However, the errors are temporary and do not accumulate since we synchronize after one set of 2^{d-1} moves using our global broadcast mechanism.

The parallel algorithm divides the circuit into a number of parts equal to the number of processors in the hypercube = 2^d. Two forms of partitioning have been proposed. In the grid partitioning scheme, the circuit is divided into rectangular regions by forming 2^k strips in the i-dimension and 2^{d-k} strips in the j dimension as shown in Fig. 13(a) for d=4, and k=2. The region assignment to different processors in the hypercube is performed in the following manner. The region $A(i,j)$ is assigned to the processor $G(i,j)$ using the binary reflected gray code mapping [34]. The assignment of regions in the partition in Fig. 13(a) to processors of the hypercube is shown in Fig. 13(b). The advantage of this region assignment is that processors that are assigned physically adjacent regions, $G(i,j)$ and $G(i+1,j)$ (similarly, processors $G(i,j)$ and $G(i,j+1)$) are physically adjacent, i.e., connected by direct links in the hypercube. This region assignment results in processors communicating with each other in the parallel algorithm to also be physically near each other. Since a lot of communication occurs between processors that are assigned regions that are physically near each other, this partitioning will result in more localized messages and hence improved performance. Note that in this assignment, each virtual grid corresponds to a horizontal portion of a number of rows. For example, for a standard cell circuit with 8 rows of cells, each processor in a 16-processor hypercube will be assigned one-fourth the horizontal length of two of the rows as shown in Fig. 13(a).

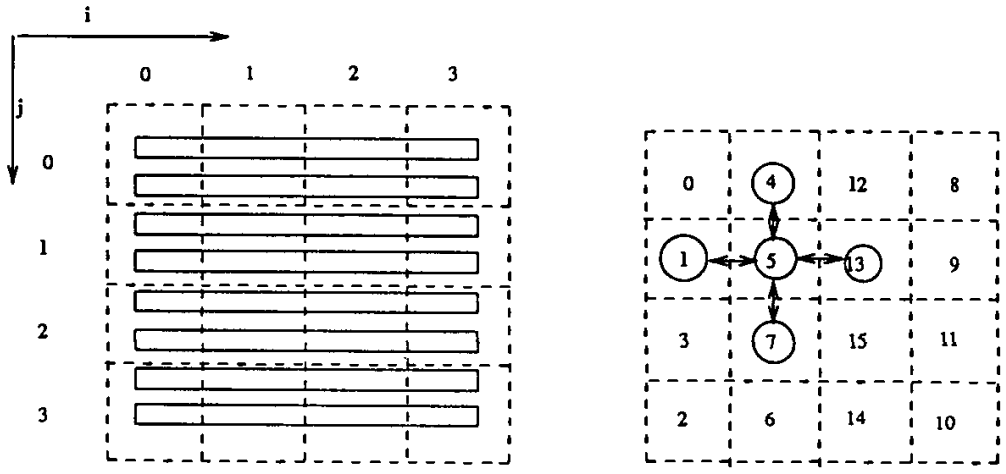


Fig. 13. Grid partitioning for 8 row circuit on 16 processor hypercube. (a) Grid partitioning. (b) Region assignment to processors.

A second partitioning scheme was proposed where each processor of the hypercube gets an entire row or set of rows of the standard cell layout. The rows are mapped using a binary reflected gray code to the processors of the hypercube as shown in Fig. 11. Row partitioning cannot achieve as fine a data granularity as grid partitioning, because the sub-area corresponding to a row of cells can be allocated to only one processor. The primary benefit of row partitioning is that the entire

cost of any move-type can be computed solely on the basis of local information, along with its partner node as will be discussed later.

The cost function used in this algorithm is identical to that used in TimberWolf3.2 [15], and consists of three subcosts: 1) estimated wire length using bounding boxes, 2) non-ideal row length penalty, and 3) overlapping cell-area penalty.

Since there is no concept of a shared database on a distributed memory machine, a distributed data structure is used to support the parallel annealing algorithm. In the grid wise partitioning, each processor contains the following information to aid in the computation of the cost function in parallel among processors in the hypercube: (1) A list of cells currently assigned to this processor along with the following information for each cell: (2) The width of the cell; (3) The (x, y) coordinate location at which the center of the cell is currently placed; (4) A list of nets to which this cell is connected; (5) For each net listed in (4), a list of other cells, to which the net is connected, along with the (x, y) pin location(s) within these cells; (6) A list of (x, y) locations and widths of all cells that are assigned to processors that are adjacent in the two dimensions of the hypercube corresponding to the East-West nearest neighbors in the physical area map.

In the row-wise partitioning, each processor maintains a list of cells currently assigned to this processor along with net-list information necessary to compute the bounding-box portion of the cost function. Note that the data structure need not contain any information about the cells in the east and west neighbor processors for every processor.

Pairs of processors cooperate to perform two kinds of cell moves — cell displacements and pairwise move exchanges. The ratio of displacements to exchanges is maintained at approximately 5:1 as used by TimberWolf3.2. One processor assumes the role of Master, the other of Slave. The relative Master/Slave relationship between any two processors alternates in time to avoid load-imbalance in cell complements. The Master determines the type of move that will be made and informs the Slave. Hence, this approach also uses parallelism. This is where we apply parallelism within a move evaluation (strategy *A* characterized in the introduction to parallel annealing approaches). Theoretically two processors cooperating can perform a cell move in half the time of a single processor, however the precedence of computational steps in resolving a move does not always allow Master and Slave to operate concurrently. There are two sub-classes of moves for both displacements and exchanges, or four move types in total: (1) Intra-processor Cell Displacement; (2) Intra-processor Cell Exchange; (3) Inter-processor Cell Displacement; and (4) Inter-processor Cell Exchange.

During an intra-processor cell displacement, the Master displaces a single cell to another location within its allocated chip area. The candidate location is chosen randomly from within a range-limiting rectangle centered upon the cells current location. During an intra-processor cell exchange move, the Master selects two candidate cells and exchanges their position. The change in cost is calculated entirely by the Master, as is the decision to accept the move. If the bounding box created by

the two candidate cells exceeds the range-limiting window in either dimension the exchange is rejected. During an inter-processor cell displacement, the Master selects the candidate cell, computes the effect of its loss, and sends a copy of the cell to the Slave. The Slave picks a new location from the area created by the intersection of the Slave's sub-area and the range-limiting box centered on the cell's previous location. Accounting for the Master's loss, the Slave computes the total cost of accepting the move, and decides accordingly. During an inter-processor cell exchange move, Master and Slave both select random cells, and both compute partial exchange costs. The Slave informs the Master of its partial cost-change calculations and the Master then makes the decision to accept the move. As in intra-processor exchanges, the move is rejected outright if the two cells are too far apart.

Processors pair up to perform one of the four types of cell moves described earlier. In the grid-wise partitioning, the processors with which each processor pairs up are governed by the direct links of the hypercube as shown in Fig. 13(b) for example for processor 5. The row-wise mapping using the binary reflected gray codes satisfies the property of processor proximity, while providing a more geographically uniform move selection. A mapping corresponding to a binary reflected gray-code sequence has the additional property that two processors with node numbers P_i and $P_{i+/-2j}$ will be separated by at most two link-hops for $0 < j < P/2$ [34]. An example of this mapping, and the pairwise communication that is possible with such a mapping, is shown in Fig. 11(b) for processor 2 in an 8-processor hypercube. Arrows originating in the row allocated to processor $n = 2$ indicate pairwise move resolution with other processors $n + / - 2j$, $0 < j < 2$. This provides geographic uniformity for hypercubes of any size.

Once the cells have been moved to new locations, these updated locations have to be sent to all processors so that they can update all net and pin information effected by the move. Two schemes for broadcasting have been implemented, one using ring based broadcast the other using a spanning tree based broadcast. The latter is more complicated but is more efficient in that the broadcasting.

As mentioned earlier, in any simulated annealing algorithm, four important criteria are the choice of the initial temperature, the equilibrium detection condition at a particular temperature, the rate of decrease of the temperature, and the frozen or termination condition. Most previously reported implementations of parallel simulated annealing use a fixed temperature sequence derived empirically [30, 36, 38, 40–42]. Huang *et al.* have proposed an adaptive cooling schedule for serial simulated annealing algorithms based on the characteristics of the cost distribution and the annealing curve itself [13]. The schedule relies on dynamic "equilibrium detection" to signal the appropriate point to lower the annealing temperature. The overhead in equilibrium detection is slight, and results in far fewer move attempts at high temperature than at low temperature, thereby reducing total CPU time. Installation of Huang's schedule in a version of TimberWolf was reported to have yielded substantial savings in overall execution time with no significant change in final placement quality. Huang's schedule was implemented in the parallel algorithm

that allowed one parallel move-set between updates. Next an error control mechanism was integrated into the annealing schedule for the parallel implementation.

In parallel simulated annealing algorithms that move multiple cells independently and simultaneously, the error in the cost function is defined to be the difference between the *real* change in cost from initial to final configurations, and the *estimated* change in cost equal to the sum of locally perceived changes in cost at each processor. If C_i is the exact cost of the initial configuration, C_f the exact cost of the new configuration, and ΔC_j the perceived change in cost computed locally at the $1 \leq j \leq P/2$ processor pairs that evaluated moves in parallel, then

$$C_i + \sum_{j=1}^{P/2} \Delta C_j = C_f + Error$$

When not written in the sum as above, consider ΔC_j to be the perceived cost change at an arbitrary processor pair. The ΔC_j of an unaccepted cell-move is zero. Unaccepted moves may have also experienced error during evaluation, but this quantity is impossible to measure directly, so average error is only sampled from accepted moves. Plainly error is due to inaccurate ΔC_j costs computed locally when evaluating potential moves. We will now try to derive an expression for the errors accumulated after several parallel moves with no intervening cell-position update. Clearly the amount of misinformation will increase with each accepted move. However if this expensive synchronizing update could be reduced in frequency, overall execution time would be significantly improved, especially at low temperature. We call such a series of parallel moves a **parallel move sequence**, or simply a sequence.

A move sequence is illustrated schematically in Fig. 14. The initial placement configuration is $S0$, the final placement configuration SN. The exact cost of states $S0$ and SN are known because the algorithm synchronizes and updates at the start of every sequence. The exact costs of states $S1$ through $S(N-1)$ are not known. However, a pseudo-cost for state $S1$ can be computed by adding the total perceived parallel move cost-change (sum of ΔC_j's) to the original cost C_0. In this manner pseudo-costs are determined for states $S2$, $S3$ up to $S(N-1)$. The pseudo-costs will become increasingly inaccurate further along the sequence as a result of increasing error. Of course the total perceived cost-change between pseudo-states cannot be determined without synchronization, so pseudo-costs are not computed until after synchronization at the end of the sequence. These pseudo-costs are used as cost samples to build a cost distribution. In Huang's uniprocessor adaptive schedule, exact cost after every move is known (no error) implicitly, and an exact cost distribution can be built. We are creating a cost distribution with error for the parallel algorithm.

The error of a sequence of length N is similar to the case for just one parallel move.

$$C_i + \sum_{k=1}^{N} \sum_{j=1}^{P/2} \Delta C_{kj} = C_f + Error\,.$$

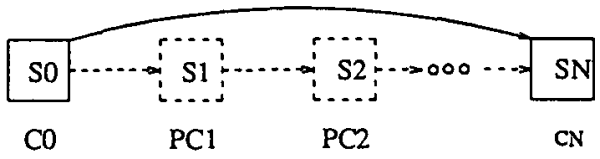


Fig. 14. Sequence of states permuted with parallel moves.

Where ΔC_{kj} is the real cost change at step k in the sequence at pair j. Again average error is just the total error divided by the number of accepted moves. As the sequence length grows, so does the amount of misinformation on cell positioning, hence average error increases.

Banerjee *et al.* have developed a new approach for controlling the cumulative error called **Adaptive Sequence-Length Control** [33, 39]. The parallel algorithm dynamically extends and contracts the parallel move sequence length constrained by a bound on allowable error derived shortly. The approach is based on finding an upper bound on the maximum permissible average error at a particular temperature. By adjusting the sequence length dynamically after each sequence the average error can be limited to a specific range. Such an approach is based on the characteristics of the move-acceptance curve. The composite move acceptance rate curve is nearly continuous and similar to the so called "annealing curve" of cost versus logarithm of temperature. The acceptance rate P has two components:

$$P = Prob\{move\ accepted|\Delta C > 0\} \cdot Prob\{\Delta C > 0\} + Prob\{move\ accepted|\Delta C < 0\} \cdot Prob\{\Delta C < 0\}$$

where ΔC is the change in cost such a move would produce. The procedure developed by Metropolis [43] rewrites the probability of accepting a "good" move ($\Delta C < 0$) as unity and that of a "bad" move as ($e^{-\Delta C/T}$). Hence,

$$P = e^{-\Delta C/T} \cdot Prob\{\Delta C > 0\} + Prob\{\Delta C < 0\}$$

In the presence of error the composite acceptance rate changes slightly, however the probability of generating good or bad moves is invariant with respect to error:

$$P_E = e^{-(\Delta C \pm E)/T} \cdot P\{\Delta C > 0\} + P\{\Delta C < 0\}$$

The approach then is to bound the magnitude of allowable error so that the "normal" composite acceptance rate is not unduly affected. To bound the acceptance rate with error P_E to within 5% of normal, i.e.,

$$\frac{P - P_E}{P} \leq 0.05$$

one can find a bound on the magnitude of error -

$$E \leq -T/\ln(1 - 0.05) \approx T/21$$

If the average error computed after a sequence is higher than $(T/21)$, the sequence length is reduced commensurate with that excess. If average error is very low $(T/42)$ then sequence length is increased slowly. Experimentally it has been found that a 5% deviation in composite acceptance maintained convergence of a large set of test circuits — it is not clear what maximum variation in acceptance rate can be tolerated and still ensure convergence. However even very tight bounds on acceptance rate allow very long sequence lengths, improving performance significantly.

The authors have reported results of the parallel algorithms implemented on an Intel iPSC/2 hypercube using both row and grid partitioning on a variety of benchmark circuits. They have reported speedups of about 12 on 16 processors with comparable quality of placement results with respect to the serial algorithms.

Rose *et al.* [40] have reported a different parallel algorithm for placement which involves a combination of two different algorithms suitable for execution on distributed memory message passing multiprocessors. Two different approaches are presented, one to replace the high temperature portion of annealing, and the other to speedup the low temperature region. The first step of the approach is to divide up the search space. A min-cut based algorithm is used to recursively subdivide a placement while minimizing the number of wires crossing a division line. A constructive initial partitioning step was introduced to aid the iterative improvement for min-cut based placement.

The second part of the algorithm uses simulated annealing for the lower temperature region which begins with an interim placement. This placement is divided up geographically into subareas. The subareas and the cells contained in those areas are assigned to separate processors. The subareas were kept are square as possible. Each processor then generates simulated annealing style moves, in parallel, for the cells which it is assigned and tests those moves for acceptance. If a move is accepted, then the accepting processor transmits the move to other processors so that they can maintain a consistent view of the cell positions. Some details about the parallel algorithm are mentioned below.

As processors generate moves in parallel, there are two choices for synchronization. Each processor can either stop after every move and wait for the others to finish (the synchronous case), or continuously generate moves without regard to state of other processors, hence the asynchronous case allows the processors to run faster. The authors used the asynchronous case in their algorithm.

The database containing the cell locations in a distributed memory multiprocessor can be kept separately or in a central location. A central database makes it easier to update and maintain consistency, however it becomes the bottleneck. The authors therefore implemented a version of separate databases in their parallel algorithm. Given that each processor maintains its own separate database, the issue is what happens if a processor accepts a move. In a Full-broadcast mode, as soon as a processor accepts a move, it broadcasts the move to all other processors. Such a strategy generates a lot of messages, and a lot of time is wasted in updating

the database. Instead the researchers proposed a Need-to-Know based broadcast where a processor only receives an update message if it contains a cell connected to a moved cell, or it contains at least one cell less than the range window distance away to calculate the overlap penalties or it contains a row whose total width has changed due to a move (to calculate the row overlap penalties).

The types of moves used in this implementation were (1) single cell displacements within and across processor boundaries (2) two cell exchanges within a processor boundary. From their experiments they noted that not allowing the cell exchanges across processor boundaries did not hamper the convergence of the algorithm.

The authors implemented their parallel algorithm on a 6 processor experimental message passing multiprocessor and measured speedups of about 4 on 5 processors.

3.5.3. Massively parallel SIMD algorithms

Two groups of researchers have reported massively parallel algorithms on SIMD machines for placement using simulated annealing [41, 42]. Both groups of researchers used a Connection Machine to implement their parallel algorithms. A Connection machine of 16,000 processors was configured as a two dimensional grid. The parallel algorithm was based on the Timberwolf sequential algorithm for standard cell placement. A single global description of the state is maintained across all the processors comprising of two data structures, one for nets and the other for cells.

Viewing the connection as an array of processors, the data structure for each cell is stored on a sequence of adjacent processors in this array. The head processor in the sequence contains general information about the cell, while the second, third and fourth processors contain position information. After the fourth processor comes a processor for each terminal on the cell, which contains the position of the terminal with respect to the center of the cell. A sequence of terminals is used to represent each net. The head processor contains general information about the net, while the processors that follow contain the absolute position of each terminal on the net, one terminal per processor.

The algorithm begins by selecting approximately half the head cell's processors. These processors consider moving their cells to a new location by either an exchange or a displacement. Next, all processors simultaneously compute the change in the cost function associated with the current move, each move assuming that the other cells are fixed.

The algorithm computes the absolute position of each terminal from the current absolute position of the cell and the relative position of the terminal. It then stores that position information in the processor associated with the same terminal in the net data structure. It uses the net data structure to compute the coordinates of the bounding box to approximate the wire length. Each cell structure uses the results of these calculations to decide whether to accept or reject the move and update its absolute position if necessary.

Since the algorithm stores a single description of the state across the machine, it ensures that after each iteration the state description is correct. During each iteration, however, the simultaneous movements of cells introduces an error.

Both groups of researchers reported implementations of the massively parallel algorithms on a 16,000 processor Connection Machine and the implementations were tested on a set of real benchmark cases. Typically in such massively parallel algorithms about 500-900 moves are attempted simultaneously on 16,000 processors. Despite the errors due to simultaneous moves, the algorithms converged, and obtained results within 2% of the serial simulated annealing results.

4. Parallel Placement Algorithms Using Simulated Evolution

Simulated Evolution is a novel optimization method that is based on an analogy to the natural selection process in biological environments. The biological solution to the adaptation process is the *evolution* from one generation to the next one by eliminating ill-suited constituents and keeping near-optimal ones. Every constituent of each generation must constantly prove its functionality under the current conditions in order to remain unaltered. The purpose of this process is to gradually create stable structures which are finally perfectly adapted to the given constraints. The simulated evolution heuristic has been applied to many VLSI optimization problems such as placement, routing and partitioning.

4.1. Overview of Sequential Algorithm

The simulated evolution algorithm for placement basically consists of a main loop in which three functional blocks are executed sequentially [20]. Each iteration of the loop completes one generation. Related to the placement problem, the three blocks shown in Fig. 15 can be described as follows: First, each cell in the population is subjected to *evaluation*, a phase which determines its normalized goodness, a figure of merit that reflects how well a cell is placed in its current position. Next, the *selection* procedure selects cells for replacement. For each cell separately, a trial is performed in which the cell's goodness determines its probability of survival in its old location. Cells with high goodness values are therefore less likely to be selected than ones with low goodnesses. Finally, the *allocation* procedure removes all selected cells from the placement. For each cell individually, a search is performed to find an improved location in the vicinity of its old position. A number of trial placements are evaluated and then a choice is made based on the amount of wire length reduction. When all selected cells have been replaced, the current iteration is complete and a new generation (= placement) has been formed. The simulated evolution method is stochastic in nature, and provides means of escaping local minima of the objective function on its path to the global optimum, thus following the biological model. In the following, we give a more detailed description of the basic steps in the simulated evolution algorithm.

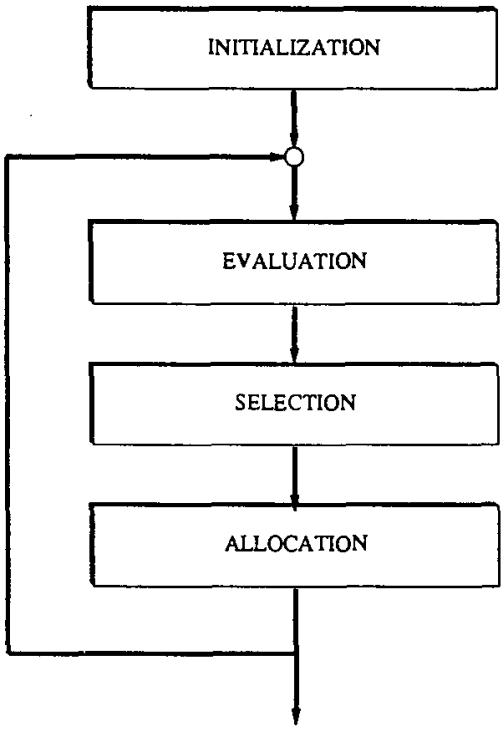


Fig. 15. Simulated evolution outline.

The first step of the iterative loop is the evaluation of the current placement. A goodness value for every cell is established. The goodness of a cell cannot be computed directly. Instead, we have to compute goodness values for all nets first. The evaluation procedure, which is done for every net separately, is illustrated in Fig. 16. The wire length of the *current net* is calculated by computing half the perimeter of its bounding box which is the smallest rectangle enclosing all pins belonging to a net. As a first approximation of an optimal relative placement, all cells of the *current net* are assumed to be placed next to each other without horizontal space in between. Row spacing is considered and included in the computation. The half-perimeter of the enclosing rectangle serves as the lower bound on the wire length of the *current net*. The ratio of its precomputed optimal wiring cost over its current wire length is determined to be the goodness measure for the net. Those steps are repeated for each net in the design. Subsequently, the goodness of each cell can be computed by taking the average of the goodness values of its *netset*. The result is then normalized in the range (0,1). Averaging over the goodness values of all cells, a global goodness of the current layout is also computed. Finally, the total wire length of the current placement is calculated. It should be noted that other goodness measures can be suitably defined.

The next step is called the selection phase which determines whether a cell will retain its current position in the next generation or if it will be scheduled for new allocation. This is done by comparing its placement value to a random number

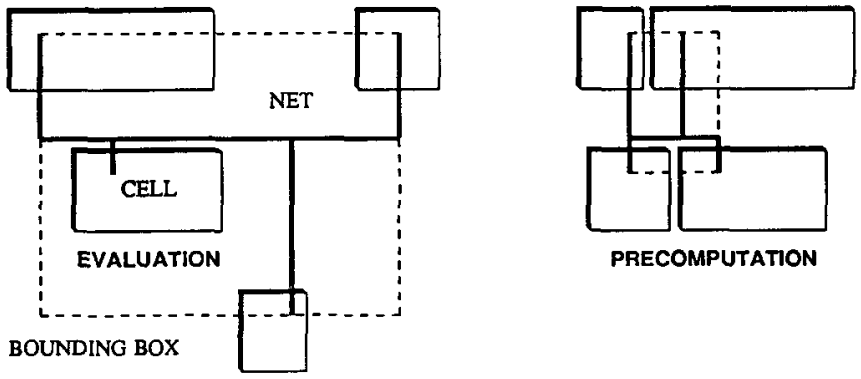


Fig. 16. Precomputation and evaluation.

generated for each cell in the range (0,1). If the goodness of the cell is larger than the random number, it will survive in its present position. Otherwise, it is placed in a queue for new allocation. By using this selection process, each cell's chance of survival at its current location is exactly equal to its placement value.

The primary operations of the allocation routine are the replacement of the selected cells into the layout, and a final realignment step to remove empty spaces and overlaps. We have found that the iterative procedure of simulated evolution can be divided into three major subphases which have unique requirements and demand appropriately suited allocation algorithms. A different allocation procedure has been proposed for each of the three phases.

During the early stages, the *sorted individual best fit* allocation method is used. All cells which were removed from the placement grid reside in the replacement queue. This queue is sorted, such that the cell with the most connections is placed first. The cell is placed into all the vacant spaces left by the replaced cells, and the goodness measures evaluated for each location. From those tentative placements, the best placement is chosen for the cell. The above procedure is repeated for the next cell in the allocation queue. The above steps are performed until all cells have been replaced. Figure 17 illustrates this process. The algorithm has a time complexity of $O(s^2)$, where s is the number of cells to be replaced which can be relatively large due to the low initial overall goodness. Therefore, the program limits this number to a maximum of a few hundred cells.

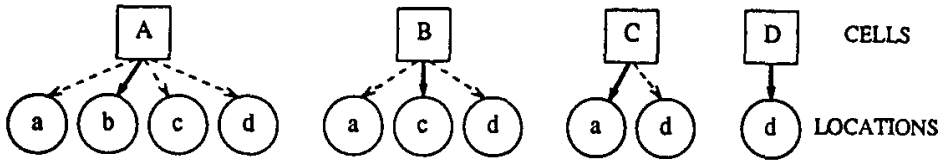


Fig. 17. Sorted individual best fit allocation.

During an intermediate stage of the algorithm, a *weighted bipartite matching algorithm* is used. The basic principle of this method is shown in Fig. 18. Before the actual replacement starts, the goodness of each cell in the queue is evaluated in every target location. This yields a table of size s^2. The matching algorithm now tries to assign each cell to a location, such that the overall wire length is minimized. This can be done optimally in $O(s^3)$ time complexity. This is an increased complexity compared with the first method; however, the number of cells s to be replaced is now much smaller, about 20–30 cells. The computation can be only as accurate as the preceding evaluation. Therefore, wires connecting unplaced cells are not considered. The overall accuracy is, however, much better than in the first method.

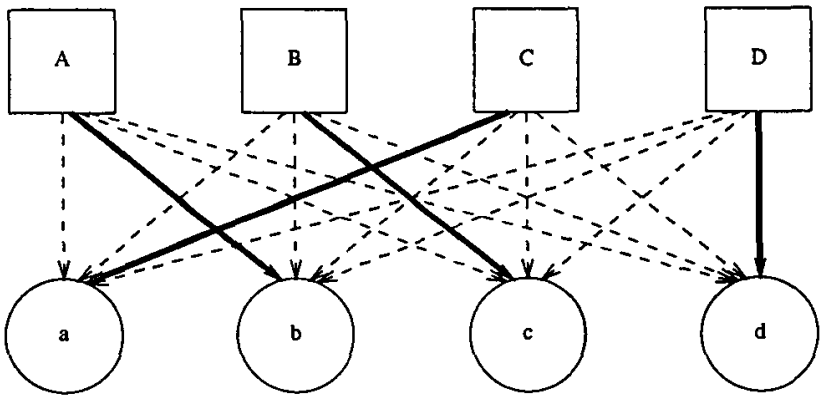


Fig. 18. Weighted bipartite matching algorithm.

The final stage of the algorithm uses a *branch-and-bound* method of exhaustive search which works by continually trying to extend a partial solution. If an extension of the current solution is not possible, a backtrack is performed to a shorter partial solution. The search tree of an optimal allocation of a small number of cells s is shown in Fig. 19. Each node in the tree corresponds to an assignment of a cell to a location. The basic idea is to calculate lower bounds on the wire length incurred at each tree node (this cost is shown in parentheses). This is done by calculating the wire lengths of the current cell's *netset*. Unplaced cells are not considered. This does not affect the accuracy of the procedure since only a lower bound needs to be computed. During the placement of the last cell in the replacement queue, all nets are accurately evaluated; therefore, the search is guaranteed to find the optimal placement. The algorithm always expands the tree node having the least cost so far and evaluates its children. The overall complexity of this method is clearly exponential; however, sorting the replacement queue can force early bounding and reduce the required CPU time. Therefore, the cells potentially incurring the highest cost due to their number of connections are placed in the front of the replacement queue. In addition, the number s of cells to be replaced is now very small, usually less than 10 cells.

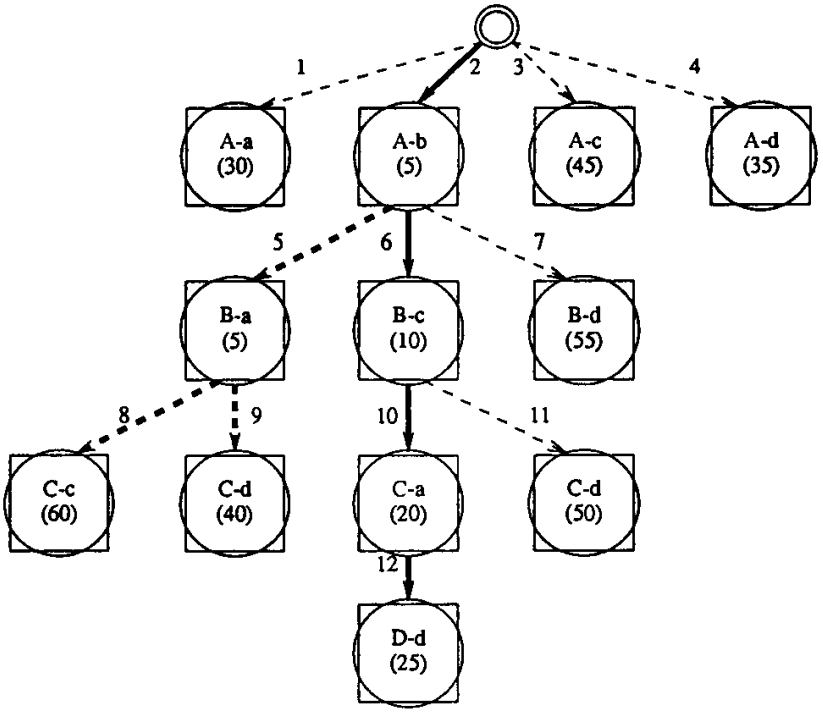


Fig. 19. Branch-and-bound search allocation.

Since the cell which is to be placed and the empty slot are generally of different widths, special consideration has to be given to the problem of possible overlaps and unused spaces. This is important because the subsequent realignment of the cells can cause the expected improvement in the total wire length to be diminished or even negated. So far, we have found that realignment after each allocation phase does not seriously disturb the convergence of the algorithm. Removing the overlaps and unused spaces between cells simplifies the program and yields legal placements at any stage of the iteration.

4.2. Distributed Memory Parallel Algorithms

A distributed memory parallel algorithm for placement using simulated evolution has been reported by Kling and Banerjee in [19]. In this section we will introduce the main concepts and summarize the results. The relatively high cost of initiating a communication on the network proves prohibitive to exploiting fine-grain parallelism on distributed memory message passing systems. However, partitioning the workload at the block-level and providing each node with a sufficient workload between the communication phases allows efficient execution.

The basic approach to parallelizing the algorithm involved assigning a subset of the physical placement grid to each processor. The partitioning is done row-wise because this greatly simplifies the adaptation of the algorithm to the parallel environment and avoids adverse effects created by unequal cell lengths. A set of rows

is mapped to each processor; it is assumed that the number of rows is much greater than the number of processors, i.e., at least 3 to 4 times greater. Two different partitioning patterns are sufficient to ensure correct operation of the algorithm on any (small) number of processors. The pattern for each processor alternates every iteration to ensure that each cell can move to any position on the grid in at most two steps, as long as the number of rows per processor is sufficiently large. The left pattern (I) shows the distribution for odd-numbered iterations and the right one (II) the partitioning for even-numbered cycles. A partitioning example for three processors is shown in Fig. 20. The different shadings represent the assignment of rows to the different processors.

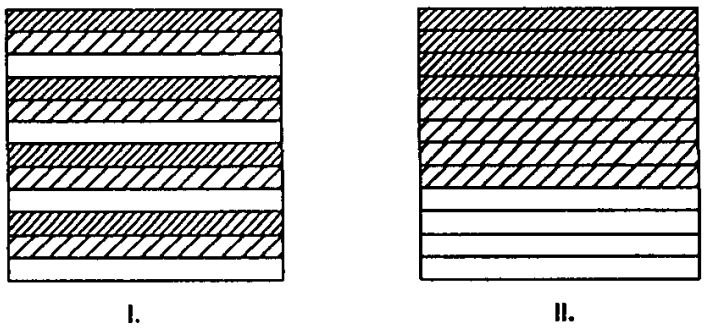


Fig. 20. Workload partitioning.

Each processor performs a complete iteration of the evolution-based algorithm, on the subset of cells assigned to it and on its assigned regions on the chip. During a subsequent communication phase, the processors update their placement information by combining all individual results to the new common placement. Then, the whole process repeats itself, starting with the assignment of new partitions. A block diagram of the distributed memory parallel algorithm is shown in Fig. 21.

Kling and Banerjee have reported an implementation of the above parallel algorithm on a network of workstations connected by a local area network. From measurements generated by placing actual circuits on a network of 1–4 workstations, the following results were obtained. For sufficiently large examples, i.e., 250–300 cells per processor, they achieved linear speedup. In this case, the processing time is at least one magnitude larger than the communication cost. In addition, the final placement quality of the distributed method was reported to be comparable to the uniprocessor case. This proved the effectiveness of our workload partitioning scheme.

It should be noted that the above parallel algorithm is effective for cases where the number of processors is much less than the number of rows. When the number of processors is equal or greater than the number of rows, a different parallel algorithm needs to be designed.

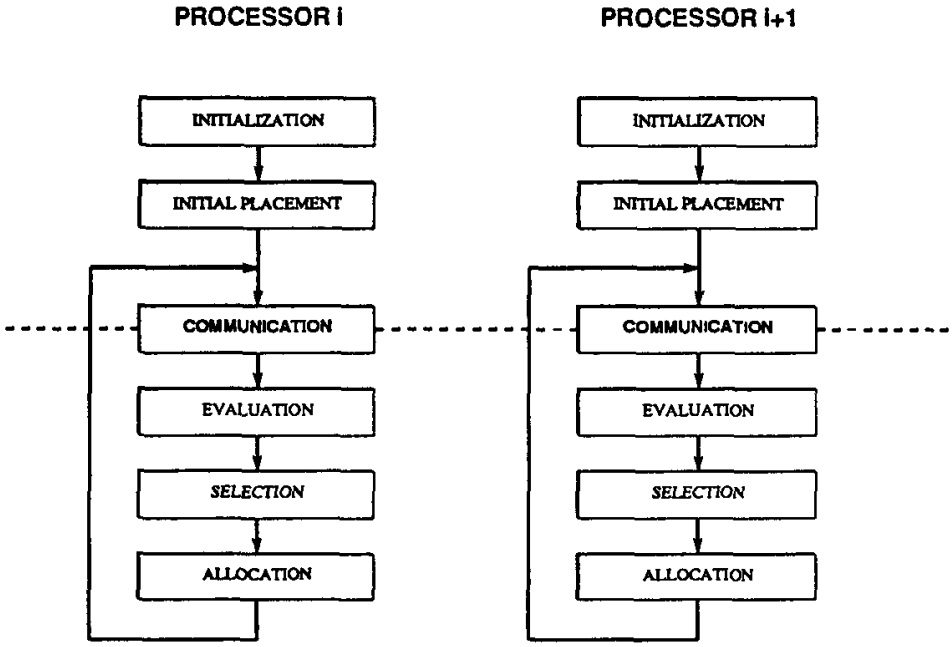


Fig. 21. Parallel simulated evolution algorithm.

In such a case, the basic strategy will be to partition the cells and associated nets among the processors. A copy of the entire chip layout will be replicated on each processor. A processor will be responsible for evaluation, selection and allocation of cells that is owned by it. After a single iteration of evaluation, selection and allocation of cells, the processors will update the global state information of the layout and resolve any inconsistencies through reallignment of the cells in the various rows. The global synchronization can be performed during the reallignment phase by as many processors as there are rows, with each processor responsible for one row. After that each row update processor will broadcast the latest correct state of the layout to all the processors. The cycle of evaluation, selection, and allocation and update will be iterated upon. Naturally, conflicts will again arise during allocation of cells to same empty slots in the layout. These conflicts will be resolved by the reallignment stage. No actual implementations using such an algorithm have been reported.

4.3. Shared Memory Parallel Algorithm

Recently, an extension to the basic simulated evolution algorithm has been proposed by Kling and Banerjee using the concepts of windowing and hierarchy [21]. The previous algorithm was extended to use hierarchical concept that is based on the partitioning of the physical layout area containing standard cells into bins. The center coordinates of each bin represent the point to which all nets of any cell inside the bin are attached. Furthermore, each bin has a capacity according to the physical area it encloses. An attempt to locate more cells into a bin than its capacity

allows incurs an overlap penalty. A sample bin layout is shown in Fig. 22.

Initially, a random placement is generated and partitioned into four bins which are filled with the respective cells they enclose. Each cell retains its individuality with respect to its net connections, size, etc. However, since all nets of a bin connect to the common center point, the length of any net entirely within a bin is set to zero. Nets connecting cells that are located in different bins are assigned wire lengths according to the physical distance between the bins. This scheme allows the *global* structure of the circuit to be optimized without interference by *local* constraints. There are no fixed positions for the cells within each bin, only the overlap penalty is enforced.

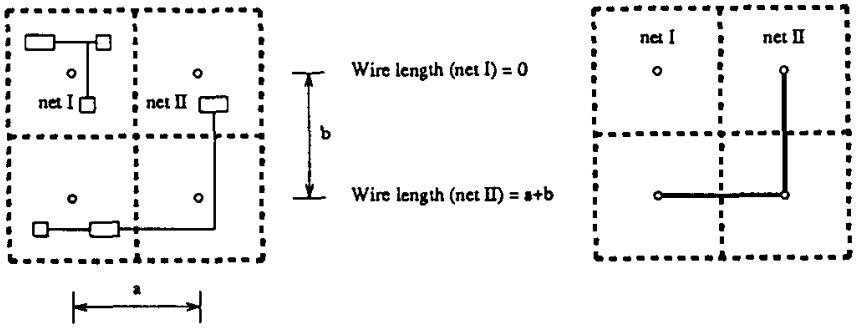


Fig. 22. Bin partitioning.

Next, the simulated evolution algorithm is executed using the bin configuration to iteratively improve the total wire length measured as described above. The method is similar to the conventional flat simulated evolution optimization procedure. The evaluation procedure determines the goodness of each cell with regard to the new wire length metric. Cells that lead to suboptimal wire lengths are penalized by low goodness values while cells with optimal nets are rewarded. The selection procedure selects a number of cells based on their current goodness values. The allocation procedure tries to replace the selected cells in improved positions by first attempting to trial place them in several different bins and then choosing an improved configuration based on the overall gain. This gain is determined by the difference in wire length between the old and the new placement and the difference in overhead penalty.

The algorithm iterates until no significant improvement can be achieved. At this point, the bins are split alternatively in either x or y dimensions as shown in Fig. 23. The partitioning routine splits one bin at a time, trying to make a reasonable decision about which cells to place into each half. The splitting of bins is a single-pass, non-iterative procedure. After all bins have been processed, their center coordinates and overlap penalties are recalculated. Subsequently, the iterative SE algorithm is restarted on the new bin configuration.

The whole process is repeated until the bins are too small to be split in either direction. At this point, a regular placement is reconstructed by placing each cell

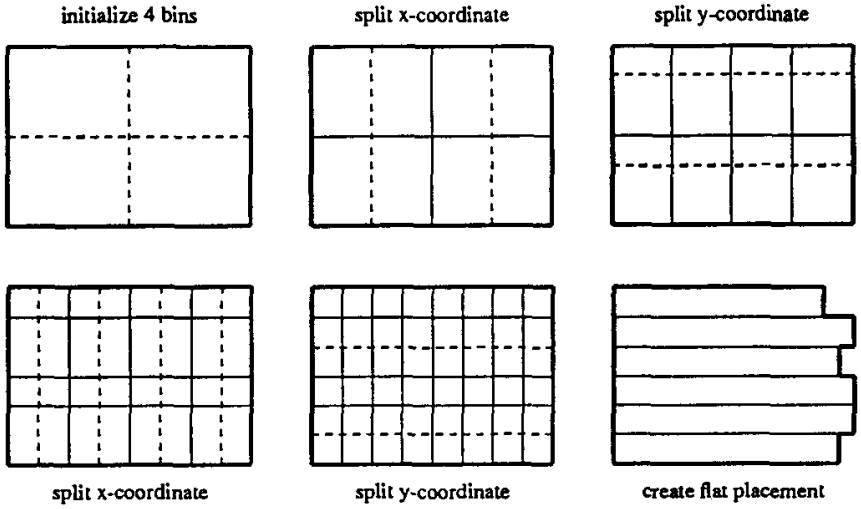


Fig. 23. Bin splitting.

within the outline of its respective bin. Finally, a standard simulated evolution optimization process is started on the flat placement.

In order to reduce computation time, only neighboring bins are considered for replacing each cell during the allocation procedure. Instead of the three deterministic allocation procedures proposed in the earlier version of evolution, a new allocation procedure was devised using windowing. A 3×3 active window is formed centered around the previous bin position of each selected cell for replacement. The cell is tentatively placed in each of the nine locations and the goodness values evaluated. From those tentative placements, a final solution is chosen using a probablistic normalization allocation function which biases the choice of making the final allocation in favor of placement locations that have high goodness values. The active window usually contains nine or fewer bins, thereby limiting the computation to a constant value independent of the total number of bins.

The windowing scheme eliminates unnecessary computations on the assumption that long range moves will occur only very rarely. In fact, the very nature of the hierarchical concept supports this paradigm since such moves should have been resolved at a previous hierarchical level. We have empirical evidence that the above assumption is indeed justified. Hierarchical programs using larger window sizes show only insignificant performance gains at best. However, the required computation time increases drastically due to the increased search space during allocation.

The main reason for the success of the hierarchical method is its inherent top-down approach to circuit partitioning. It should be noted that no predefined clustering scheme is employed, which could potentially severely limit convergence by imposing an inflexible framework. The method instead relies on the fact that the nets which are the longest in the circuit will be placed first without being constrained by the requirements of shorter nets. This effectively reduces the number

of local minima encountered by the optimization procedure, thereby reducing its likelihood of getting trapped and avoiding the overhead associated with escaping local minima.

With each splitting of the bins, the granularity of the optimization algorithm is refined and ever shorter nets are considered, thereby slowly progressing from global to local optimization. When the final flat optimization stage is reached, the remaining amount of disturbance that requires improvement is usually minimal.

Banerjee and Kling have proposed a shared memory parallel algorithm based on the hierarchical simulated evolution algorithm. The entire hierarchical algorithm is executed as a set of phases in each level in the hierarchy, which are synchronized by barriers across processors.

The evaluation procedure is parallelized as follows. A single global copy of the state of the layout, i.e., the locations of the cells in the appropriate level in the hierarchy is kept in a shared memory. The total number of cells are distributed equally among all the processors statically. It is possible to achieve good load balancing since all cells are subjected to the evaluation and selection phases at every step in the algorithm. The cost of evaluating a cell depends on the number of nets it is connected to, and for each net, the number of cells on the net. A good static load balancing strategy is to sort the cells by the number of connections and assign the cells in a cyclic manner to the processors. Namely, the cells, 1, $N+1$, $2N+1$, $3N+1$, ... goes to the first processor, the cells 2, $N+2$, $2N+2$, $3N+2$, ... to the second processor, etc.

The selection procedure is a simple random number generator call followed by a decision to select or not select the cell and is therefore easy to load balance. A barrier synchronization is performed at the end of the selection phase.

At the end of the selection phase, all the selected cells are removed from their current cell locations and placed on a queue. Processors take each cell from the queue and apply the allocation function on each cell's window of 3×3 cells, and use a stochastic procedure to accept a particular allocation. If there is a single queue of cells to be replaced, there will be conflicts for access from the shared single queue hence it may be desirable to split the queues into one queue per row of the circuit layout, and assign one or more queues per processor.

When all the cells from the queue(s) have been placed, a barrier synchronization is performed, and the cell realignment is performed on each row of the layout in parallel by different processors. The above procedure is repeated many times at a particular hierarchical level until no improvement occurs. At that point, the bins are split again and the parallel simulated evolution algorithm is performed at the next level in the hierarchy.

No actual data on the speedups of real circuits on a real multiprocessor has been reported. However, the performance of the parallel algorithm can be analyzed theoretically. The problems faced by parallel simulated annealing researchers about errors in evaluating parallel moves is not experienced by this parallel algorithm because the simulated evolution algorithm is naturally parallel. The sequential al-

gorithm for allocation ignores the placement information of the cells that are on the replacement queue. Hence the parallel algorithm also makes the same inaccuracies in the judgement as the sequential algorithm, and the result of the solution does not deteriorate with increasing number of processors. As an aside, it has been shown through a theoretical Markov Chain analysis [21] that the simulated evolution algorithm with the new allocation scheme will converge to the optimal solution given infinite time, similar to the simulated annealing algorithm.

5. Parallel Placement Algorithms Using Genetic Approaches

Genetic algorithms have been proposed as another probabilistic optimization technique that has been applied successfully to several optimization problems. Recently, researchers have applied genetic algorithms to solve problems in VLSI CAD such as placement and floorplanning.

Genetic algorithms represent and transform solutions from a problem's state space in a way that represents the mechanics of natural evolution. The subspace of solutions currently being looked at is called a population. Solutions are represented as strings of symbols. The solution strings are ranked by a function which assigns a score to the string, which is a quality measure of the string. During each iteration of a genetic algorithm, a certain fraction of the population are selected to be parents by a choice function. New solutions called offsprings are created by combining pairs of parents in such a way that each parent contributes to the information carried by its associated offspring string. This operation is referred to as crossover. From the combined set of original population and the offsprings, a set of strings equal in size to the original population is probabilistically selected to survive to the next iteration or generation. In addition to crossover, another operation called mutation periodically performs some random change within each surviving strings. Over many generations, better scoring strings representing solutions of better quality survive, and weaker solutions become extinct.

5.1.1. Overview of sequential algorithm

The above genetic algorithmic paradigm has been adapted to the standard cell placement problem by two groups of researchers, Cohoon and Paris in [22], and Shahookar and Mazumder in [23]. The string representation in both placement algorithms is as follows. The ith element within a string corresponds to the placement of the ith module in row major order among all possible placement locations in all the rows of the standard cell layout. The scoring function is a sum of the half-perimeter bounding box for each net (an estimate of the total wirelength) and a component that penalizes the usage of channels whose usage is one standard deviation more than the average (an estimate of congestion). The initial population is constructed using a combination of random placements and constructive placements using seed clustering based ideas, where after some seed modules are placed, other modules on nets connected to the placed modules are assigned to locations

in the immediate vicinity in row major order. The choice functions chooses parents to be the solutions whose quality is better than the average quality of the population. The crossover operator selects two parent strings and exchanges substrings between parent strings to create offsprings. The selection operator probabilistically favors solutions with better qualities to survive. The mutation operators selects two cells from one string and exchanges their locations. Two implementations of genetic algorithms for placement have been reported [22, 23] and have been found to produce comparable results to a basic simulated annealing algorithm in about the same runtimes.

Numerous crossover operators have been proposed by researchers. One such crossover operator was proposed in [22] will be described below. One parent is selected randomly to be the basis of the offspring, and is called the target parent. The other parent is called the passing parent. A $k \times k$ (where $k = 2$ or 3) window C of the passing parent is chosen randomly, and is copied into the corresponding window D in the target parent. In this cut-and-paste method, an illegal solution is obtained since some of the modules are repeated and other modules have no assigned locations. This solution is corrected by moving modules that are in window D but not in C to the positions of modules in window C but not in D. This operation is illustrated in Fig. 24. Other crossover operators have been proposed. The order crossover operator [23] randomly chooses a cut point in the string representation, and copies the cells from one parent to the offspring to the left of the cut point. The remaining portion of the offspring are copied from the other parent by selecting cells that were left out in the order of appearance in the second parent. In a cycle crossover, illustrated in Fig. 25, in the offspring each cell is in the same location

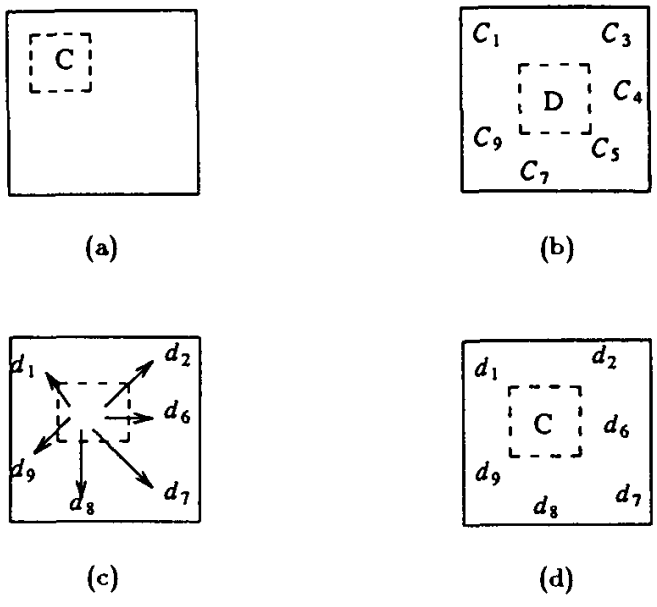


Fig. 24. Cut-and-paste crossover operator in placement.

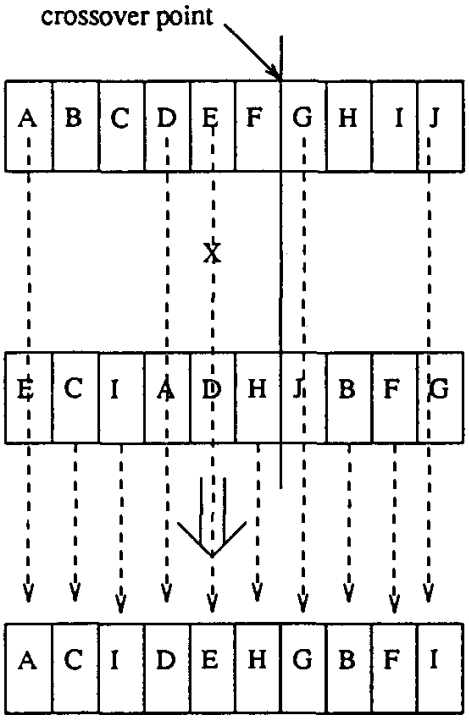


Fig. 25. Cycle crossover in placement.

of either one of the parents. Start with a cell in location 1 of parent 1, and copy to location 1 of the offspring. The cell in location 1 of parent 2 cannot appear in location 1 of the offspring, hence search for the location x in parent 1 where that cell is located, and copy that cell to location x of offspring. Now cell in location x in parent 2 cannot be copied to location x in offspring. Hence a similar search is repeated until a cycle is completed where one reaches a cell that has already been passed. The same procedure is started again from a cell from the second parent and is repeated until all the cells are assigned positions.

A typical genetic algorithm for placement would appear as follows.

```
ALGORITHM GENETIC-PLACE;
    Read net list and create initial populations of placement configurations.
    Evaluate all placement configurations in the population.
    For G Generations (Iterations) do
        Perform Crossover operation
        Perform Mutation operation
        Evaluate Placement configurations.
        Find Average fitness
        Select configurations for next generation, with fitter individuals getting
        higher probability of selection.
    end for;
END ALGORITHM;
```

Shahookar and Mazumder have proposed [23] another variation of the basic genetic placement algorithm. A genetic algorithm is characterized by the number of offsprings to be generated (crossover rate) the fraction of populations to be mutated (mutation rate). Optimal values for these parameters are obtained automatically by running another meta-genetic algorithm for optimization of those parameters.

5.1.2. Distributed memory parallel algorithm

The above genetic algorithm for placement is very easy to parallelize. Even though no actual implementations of parallel genetic algorithms for placement have been reported, it is possible to develop a parallel algorithm for placement as a direct extension of a parallel algorithm for floorplanning using genetic algorithms proposed by Cohoon *et al.* [44].

The parallel algorithm is based on the theory of punctuated equilibria for genetic algorithms [44]. A set of n solutions is assigned to each of N processors, for a total population size of $n \times N$. The set assigned to each processor is its subpopulation. The processors can be connected by a connection network such as a hypercube or a mesh that offers sufficient connectivity and small diameter to ensure adequate mixing of populations as time progresses. The overall structure of the genetic algorithm is given below.

```
ALGORITHM PARALLEL GENETIC-PLACEMENT;
Initialize;
For E iterations do
        For each processor i in parallel do
          Run Genetic Algorithm for G generations
        endfor
        For each processor i in parallel do
          For each neighbor j of i do
            Send a set of solutions S_ij from i to j
          endfor
        endfor
        For each processor i in parallel do
              Select an n element subpopulation
        endfor
endfor
END ALGORITHM;
```

The E major iterations are called epochs. During each epoch, each processor disjointly and in parallel executes a genetic algorithm on its subpopulations. After this phase, each processor sends randomly selected subsets of subpopulations of size s to each of its neighbors. Finally, each processor probabilistically selects a set of n solutions to survive using a fitness measure. The genetic algorithm code on each processor is as described earlier.

Even though no actual parallel genetic placement algorithms have been reported, it is possible to estimate the performance of such an algorithm. If the total population size is kept constant for the uniprocessor and multiple processor runs, one should expect to get similar quality of palcement results. For example, the size would be 100 for the uniprocessor, and 10 per processor for a 10 processor case. The speedups woiuld then be almost linear since the communiucation overhead is not that significant.

The above algorithm is, however, not scalable for very large number of processors, because typically the population sizes in such genetic applications has been found to optimal at around 25 for sequential implementations [23]. In a parallel implementation of the algorithm, one has two options in order to maintain near linear speedups. For a 5 processor implementation, run genetic algorithms on each processor with a subpopulation size of 5, or to run each with a subpopulation size of 25, but make the corresponding uniprocessor run with a population size of 125. Both schemes have disadvantages. For the first scheme, if one splits the subpopulations up to less than 10, the genetic nature of the algorithm would get lost and the quality would be affected. For the second scheme, one is clearly comparing with an inefficient version of genetic placement on the uniprocessor as such large population sizes are not needed.

5.1.3. Shared memory parallel algorithm

The above parallel algorithm can be easily adapted on a shared memory multiprocessor. Essentially, the entire set of solutions will be kept in global memory. However, the populations will be partitioned into subpopulations allocated to individual processors. Processors will try and acquire two solutions from within its own population subspace, and perform the crossover and mutation operators on them to generate offsprings. Next, the processors perform the choice function on the subpopulations in order to select the surviving populations. The operation of transferring subpopulations across processors will be performed by selecting a set of solutions from each processor and copying them into the solution regions of other processors. This copying needs to be performed by locking pairs of processors which will perform the exchange of their solutions. No actual implementations of a shared memory algorithm has been reported but one should expect similar linear speedups as the distributed memory implementations.

6. Parallel Placement Algorithms Using Hierarchical Decomposition

In this section we will discuss two parallel algorithms for placement, one for shared memory multiprocessors, and one for distributed memory message passing machines. Both the algorithms have the common theme of hierarchically dividing the problem and solving each subproblem in parallel among various processors. The actual decomposition approaches are different in the various algorithms.

6.1. Overview of Sequential Algorithms Using Hierarchy

Placement methods that are based on a partitioning strategy usually have a goal to minimize the number of nets crossing over the partition boundaries. The bisection (or min-cut) method [6] partitions the layout (i.e., the circuit cells) into two groups, performing cell swaps or displacements between the groups until the number of nets crossing the single boundary is minimized while maintaining a balance in the cell area of both halves. In the quadrisection method of Suaris and Kedem [45, 46], the layout is partitioned into four groups (a two-by-two array of bins) instead of two groups, and cell movements occur among any of the four groups. An extension of the bisection heuristic for the selection of the cells is applied, which minimizes the net cuts over all four boundary segments of the two-by-two bin array through the movement of the selected cells. By approaching the layout problem in two dimensions instead of one, the authors have demonstrated results much better than those attained with the use of bisection placement. At each level of the quadrisection decomposition, a portion of the layout is selected and divided into four quadrants. At level k in the decomposition, the entire layout is divided up into a $2^k \times 2^k$ array of quadrisection regions.

If a net connection from the area outside of the layout portion must enter into one of the quadrants, a *pseudo pin* is fixed in that quadrant for the net. These pseudo pins are the result of previous routing evaluations which have determined that certain nets cross into the layout portion through specific segments of the quadrisection outer boundary. The quadrant associated with the boundary segment receives the pseudo pin. According to [46], each net can be associated with a cost which is calculated as a function of the net's routing configuration for this set of quadrants. Their cost function assumes that the shortest path is always available for connecting the pins in the quadrants. The horizontal (hw) and vertical (vw) weights are used to account for differences in the costs for routing different directions, and are usually specified by the user. Banerjee and Brouwer have proposed a better cost function which determines how each net is routed and calculates each net's cost based on the routing crossings of the four boundary segments [47]. Non shortest path routes are allowed if they result in decreased costs in terms of horizontal or vertical cuts.

If a given cell c in quadrant q were to be moved to quadrant r, the nets associated with c may have to be rerouted to make the connections to the new pin in r. Furthermore, if c were the only connection for a net n in q, the connections to q for n may be removed also. These changes or reroutings of the nets cause changes in the calculated cost of the net. In order to account for the change in cost, a system of *gain tables* is used which reflects the change (gain) in cost of the nets with respect to movements of cells from one quadrant to another. A separate gain table is used for each of the twelve combinations of $q, r \in \{0, 1, 2, 3\}$ such that $q \neq r$. Since our goal is to minimize the net length and cost, a cell is selected for movement from a gain table when it has the best (smallest) cost gain.

In addition to the determination of the minimum gain cell, another important criterion in the selection of cells is the determination of whether the movement of the cell would cause an imbalance in the total area of the cells occupied by each quadrant. The sequence of selecting and moving cells is repeated until no cells can be selected for movement or when a sequence of selections of cells with gains > 0 has taken place.

6.2. Shared Memory Parallel Algorithm

Brouwer and Banerjee have proposed an algorithm for placement using hierarchical decomposition techniques similar to the quadrisection at various hierarchical levels that is suitable for parallel execution [47]. The parallel algorithm consists of a number of operations which are executed in a certain sequence at each level of the hierarchical decomposition. The operations are (1) the placement of the current set of cells, based on a variation of the quadrisection algorithm, (2) the routing of nets for the quadrisection placement, (3) the restricted global bisection of cells, and (4) the *two-by-N* routing of the nets in the bisection. Each operation intimately depends on the results of the other operations.

(1) *Quadrisection operation.* The algorithm uses a quadrisection based placement algorithm to place cells into four quadrants at a particular level in the hierarchy with a slightly different route-based cost function that pays attention to how the nets that are cut by the partitioning are routed.

(2) *Quadrisection Routing.* At the end of the quadrisection based placement operation, the cells of the portion of the layout have been placed in one of the four quadrants while minimizing the net crossings over the boundaries between the quadrants. A quadrisection routing operation can then be used to verify and lock in place the routing of the nets across the four boundary segments. A single iteration of the routing algorithm presented in [48] may be used to determine the routing of the nets across the four quadrant boundaries. Since the route-based cost function is used in the quadrisection placement algorithm, it is necessary to know the routing of the nets in the quadrisection region before quadrisection can take place. The routing must be based on the current placement of the cells at the beginning of quadrisection. Thus, one iteration of the two-by-two routing algorithm will be performed before as well as after the quadrisection cell placement when the route-based cost function is used.

(3) *Restricted Global Bisection.* The initial placement of cells in the partitioning for quadrisection is performed using a method called Restricted Global Bisection. The bisection is performed separately in the X-dimension and the Y-dimension. The X-dimension bisection consists of the set of cells between the coordinates x_{lo} and x_{hi} and the bottom and top borders of the layout. The values for x_{lo} and x_{hi} are determined by the quadrisections at the previous level in the hierarchical decomposition. The vertical lines separating the quadrisection regions and the vertical lines which cut down the middle of a column of quadrisection regions are

used as the domain of the values of x_{lo} and x_{hi}, thus giving 2^k separate X-dimension bisection regions, where k is the current hierarchy level. Given x_{lo} and x_{hi} for a bisection, the set of cells is then partitioned into two groups, separated by a line (x_{mid}) halfway between x_{lo} and x_{hi}. The bisection region is further divided up vertically, with the number of partitions $PRT_{bsect} = 2^k$. Throughout the bisection algorithm, the cells are restricted to horizontal movements only. Thus, every cell stays in the same vertical partition and each cell's y-coordinate remains untouched.

Alternately, the Y-dimension restricted global bisection algorithm partitions the layout into horizontal strips the width of the layout area. By applying the cost function horizontally, we reduce the demand of the nets for a high number of feedthroughs and route each net in as few channels as possible. Since the X- and Y-bisection algorithms exclusively alter the $x-$ and $y-$coordinates of the cells (respectively), they are independent of each other, and the two dimensions can be evaluated simultaneously. Following both evaluations, the cells have been preplaced in one of the quadrants of the quadrisection to be involved in the current level of the decomposition.

(4) *Two-by-N Routing.* Following the bisection operation, a two-by-N routing of each bisection region is performed, using the Burstein and Pelavin algorithm [48] with the parallel algorithm implementation ideas presented by Brouwer and Banerjee in [49]. The goal of the routing is to determine the sets of nets crossing each half of the partition lines running perpendicular to the the bisection line at x_{mid} (or y_{mid}). Although the bisection routing was introduced as immediately following the bisection placement, it is necessary to perform a bisection routing immediately after the quadrisection placement also. The two-by-N bisection routing following the quadrisection placement is necessary not only because the balancing of cell areas may change the best routing between quadrisections evaluated in parallel, but also to optimize the routing connections following movements of cells among the quadrants. Thus, at each level of the hierarchical decomposition, the bisection routing algorithm is effectively applied twice.

The overall flow of the parallel algorithm using all four placement and routing steps discussed above is illustrated in Fig. 26. In this figure, each operation performed at each level of the hierarchical decomposition is denoted by a set of circles between a pair of horizontal dashed lines intersecting the appropriate column. The circles represent instances of the operations to be performed on a portion of the layout. For example, in the quadrisection placement column, one circle at hierarchy Level 0 represents a quadrisection covering the entire layout. Four circles at Level 1 represent the four quadrisections, each covering one-fourth of the layout.

At each level of the decomposition, the cells are initially placed using the global X- and Y-bisection placement algorithm. This is immediately followed by a two-by-N routing of the same regions to determine the net crossings for the boundaries of each quadrisection placement region on that level. Next, a two-by-two routing of each quadrisection placement region is performed, taking into account the nets crossing through and ending in the region, to set up the current routing configura-

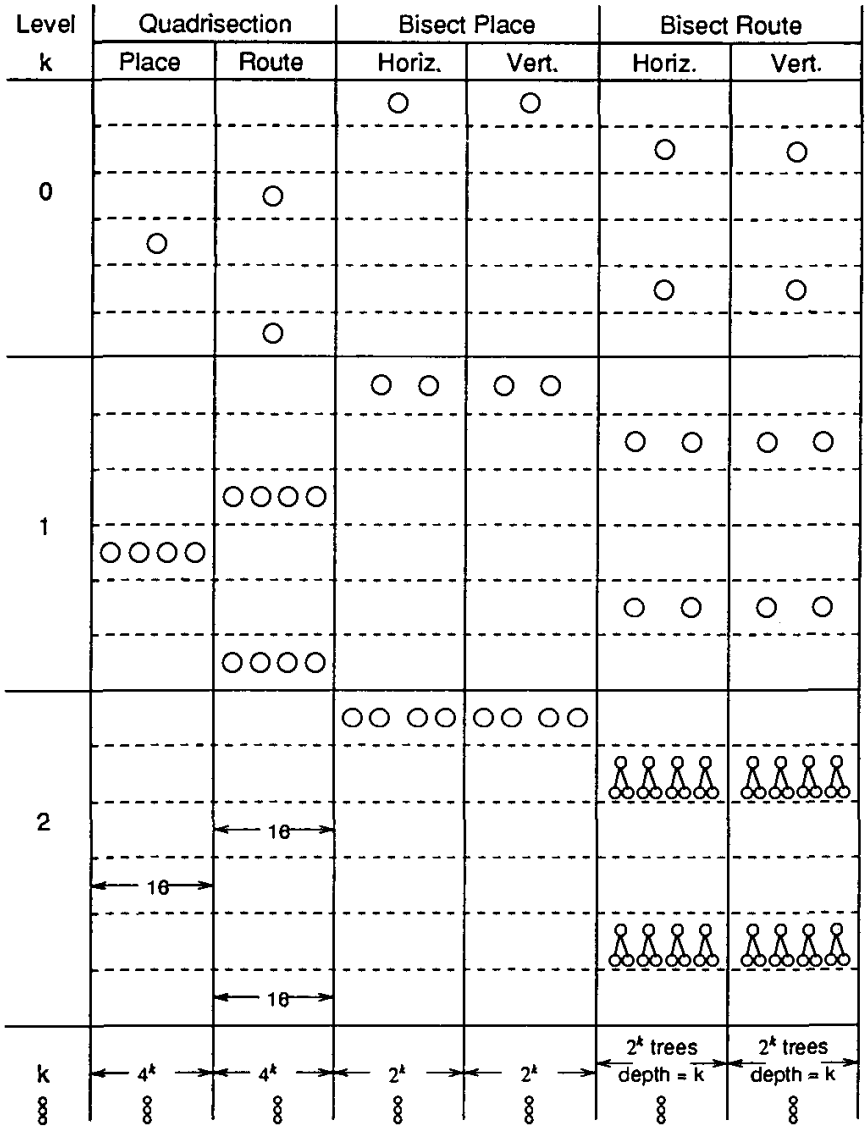


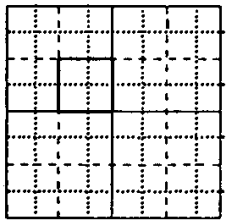
Fig. 26. Placement and routing decomposition.

tion for each net to be used in the quadrisection cost function. The quadrisection placement algorithm is then used to improve the current locations of the cells (provided by the previous bisection placements). Since the previous two-by-N routing may need to be modified due to movements of the cells inside the quadrisection region, the two-by-N routing algorithm is repeated. Finally in the hierarchy level, a two-by-two routing is applied to each quadrisection region to fix the crossing locations of the nets on the four cut lines ($A - D$) separating the four quadrants.

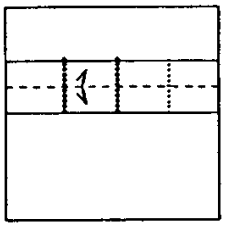
An example showing the operations at Level 2 in the decomposition for one region of the layout is shown in Fig. 27. In Fig. 27(a), the region under consideration is the square outlined in bold. The dashed lines denote the boundaries of the bisection region for Level 2. The dotted lines represent the internal axis lines of the bisections and quadrisections for Level 2. At Level 3 the dotted lines would represent

the bisection and quadrisection borders. Figures 27(b) and (c) show the horizontal and vertical bisections, respectively. The cells being displaced must remain between the pairs of bold dotted lines. Figures 27(d) and (e) show the horizontal and vertical 2×4 routing of the bisection regions. Each 2×4 routing requires the solution of three 2×2 routing instances. Figure 27(f) shows the quadrisection routing that is necessary before route-based quadrisection can be performed (Fig. 27(g)). Figures 27(h), (i), and (j) show the repetition of the routing operations performed earlier.

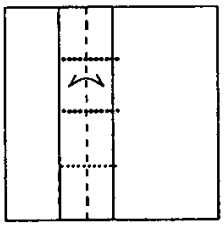
The parallel algorithm for placement has been implemented on an Encore 510 Multimax shared memory multiprocessor. The authors have reported on the results of the placement qualities and speedups on various benchmark circuits. The



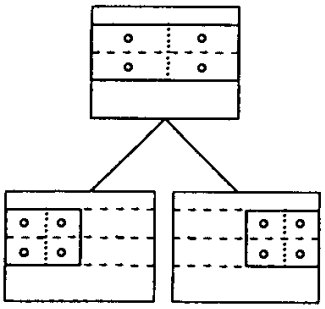
(a) Region under consideration



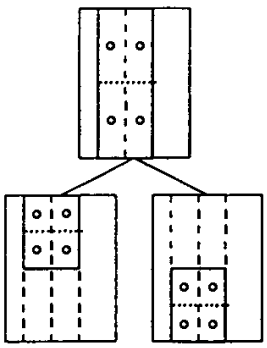
(b) Horiz. bisection placement



(c) Vert. bisection placement

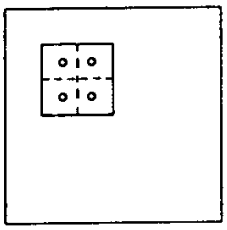


(d) Horiz. bisection routing

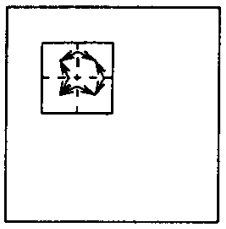


(e) Vert. bisection routing

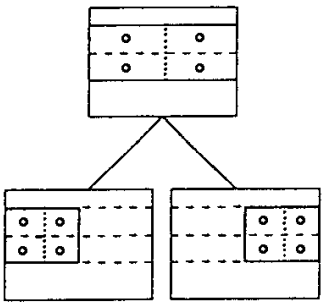
Fig. 27. Parallel decomposition example.



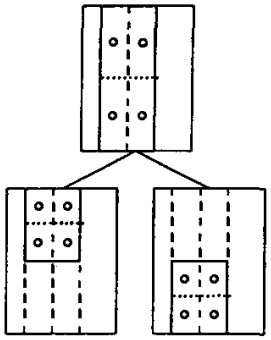
(f) Quadrisection routing



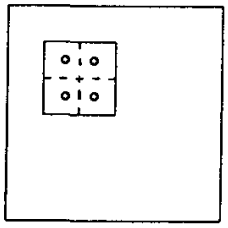
(g) Quadrisection placement



(h) Horiz. bisection routing



(i) Vert. bisection routing



(j) Quadrisection routing

Fig. 27. (*Continued*).

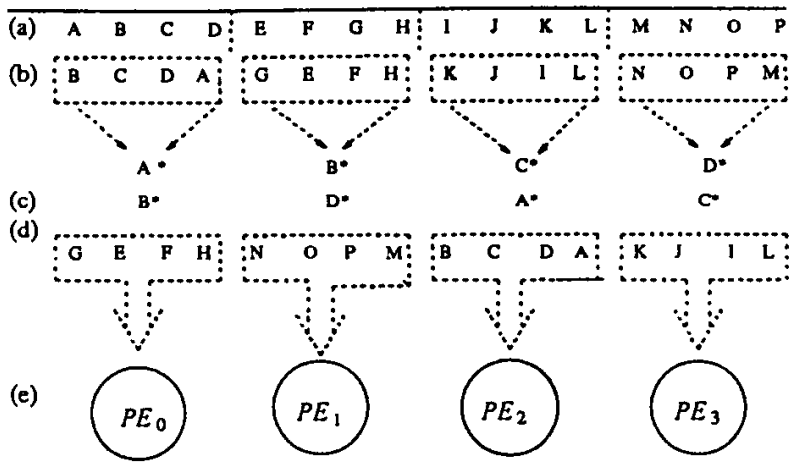
qualities of the circuits produced were equivalent to those obtained by the sequential quadrisection algorithm of Suaris and Kedem [45]. A point to note is that the quality of the circuits produced by the parallel algorithm was identical for different number of processors. This should be contrasted to the varying qualities of results produced in parallel algorithms for placement based on iterative improvement. The observed speedups varied from about 3 to 4 on 8 processors for the various circuits.

6.3. Distributed Memory Parallel Algorithm

A different parallel algorithm for module placement using a divide and conquer approach has been proposed by Ravikumar and Sastry [50]. It is based on the simple placement model of placing a set of equal sized modules on a two dimensional array of slots. The connections between the modules are specified in terms of a

connectivity matrix C, where the element $c_{ij} = c_{ji}$ = the number of connections between modules i and j, asssuming only two terminal nets.

The basic approach of the parallel algorithm involves divide and conquer. DAC placement proceeds by breaking up the connectivity matrix C into smaller pieces. Each submatrix corresponds to a cluster of modules. A cluster represents a placement problem of size small enough to be placed optimally using exhaustive or branch and bound techniques. Following the local placement phase, the clusters are shrunk into hypothetical modules, obtaining a placement problem whose size is the same as the number of clusters, l. If l is small enough, the clusters are placed optimally. Otherwise, the divide and conquer procedure is recursively applied. Figure 28 illustrates the procedure DAC for $N = 16$, and $m - 4$.



(a): 4-way partition of 16 modules (b): local placement phase
(c): Shrinking the C-matrix (d): cluster placement
(e): Final placement

Fig. 28. Example application of DAC placement.

The above basic concept is used to develop a parallel placement algorithm for a hypercube.

ALGORITHM PARALLEL-DIVIDE-AND-CONQUER;

1. The N modules to be placed are divided into $\lceil \frac{N}{m} \rceil$ clusters and distributed equally among the processors.
2. Each processor works on $\lceil \frac{N}{m.P} \rceil$ local placements each of size m. The local placement is done by semi-enumerative techniques. At the end of the placement, each processor computes a cost of the local placement, L_i.
3. Each processor i computes the collapsed connectivity by collapsing the clusters into hypothetical modules.
4. The problem of placing N/m hypothetical modules among P processors is done by a distributed placement algorithm which minimizes the inter-

cluster cost by using a parallel iterative improvement algorithm similar to the algorithm of Ueda *et al.* discussed earlier [28].

5. The global cost GC is calculated by summing up the local costs L_i and the Intercluster costs.
6. A master processor decides to accept or reject the current placement based on the value of the global cost by comparing the value of the GC for the previous placement. A greedy criterion or Metropolis criterion may be used to accept/reject the move.
7. The last step is a placement perturbation. Two types of perturbations are used. The first is an intracluster perturbation, where modules within a cluster are subjected to a random change. The second form of perturbation is among modules belonging to different clusters. This is done by partitioning the hypercube into two half cubes, and letting clusters that exist across directly connected processors in the two half cubes to interact. This reduces interprocessor communication overhead.
8. The steps 2 through 7 form one iteration of the DAC placement. This is repeated a certain number of times which is a user parameter.

END ALGORITHM;

The performance of the parallel DAC algorithm has been analyzed theoretically by the researchers. The speedup S on P processors for N modules has been determined to be

$$S = \frac{1}{1/P + \log(P)/N}$$

which converges to P for large values of N. However, the authors have not performed any studies with regard to the quality of the placements produced by the algorithm.

6.4. Combination of Approaches to Parallel Placement

It is often a common practice in sequential implementations of good placement algorithms to combine a variety of approaches. For example, constructive approaches such as min-cut based placement are often followed by iterative improvement based placement algorithms. Until now we have discussed parallel algorithms for placement by parallelizing the placement algorithm in one particular way. Many researchers have actually proposed parallel placement implementations which combine a set of approaches to get the best results.

The first such work was reported by Kravitz and Rutenbar [30] who combined two forms of parallel annealing approaches. The authors have proposed the use of combining multiple strategies to exploit the best feature of each scheme at different temperatures. (1) Move decomposition at high temperatures (2) Parallel Moves at low temperatures. At low temperatures since relatively small fraction of moves get accepted, the parallel moves strategy with acceptance of only a single moves gives a lot of parallelism. At high temperatures, a large number of potential good moves are not accepted so it gives poor speedups. The move decomposition strategy

gives good speedups at high temperatures. Determination of the optimum crossover temperature at which to switch strategies is a hard problem. The approach used was to estimate the speedup of the Static Function Move acceleration as a function of temperature and compare it to the speedup obtained through the Parallel Moves Strategy. They switched strategies at the point that the speedup of the parallel moves strategy was higher than the move acceleration strategy.

Rose *et al.* [40] have reported another parallel implementation for placement which involves a combination of two different algorithms: one to replace the high temperature portion of annealing, and the other to speedup the low temperature region. In the regular simulated annealing algorithm at high temperatures, the general search space being investigated changes rapidly due to the high acceptance ratio and the large scale of moves. The result of the high temperature phase is a coarse placement that assigns each cell to a general area. An alternative to sequentially searching a number of coarse placements is to generate and investigate different coarse placements in parallel. This is the basic idea of what the authors have termed Heuristic Spanning.

Essentially a heuristic algorithm is used to generate a number of grossly different but plausible placements at the same time on different processors. These are evaluated by another heuristic procedure to produce an interim goodness measure. The best one of the interim placements is chosen to be annealed further. A key point is that the heuristic algorithm runs much faster than simulated annealing. The first step of the Heuristic Spanning approach is to divide up the search space. A min-cut based algorithm is used to recursively subdivide a placement while minimizing the number of wires crossing a division line. A constructive initial partitioning step is introduced to aid the iterative improvement for min-cut based placement. Experience has shown that using different seeds has a marked effect on the quality of the final placement. The idea of Heuristic Spanning is to let multiple processors start from different seeds that are as far apart from each other as possible. The second step is of the Heuristic Spanning approach is to choose one of the interim placements to be annealed further at low temperatures. The authors have found experimentally that choosing the interim placement with the lowest interim cost function generates the lowest final cost function at the end of annealing as well.

The final step is the low temperature annealing. The choice of the starting temperature of the annealing process is difficult. The following method has been recommended by the authors. Determine the cost of the interim placement. In a regular simulated annealing run, obtain a table of cost function versus temperature. Determine which temperature of the simulated annealing run has the closest cost function to the interim cost function. Choose that temperature as the starting temperature. A problem with this strategy is that one needs to perform a simulated annealing run for the circuit to determine the table of cost versus temperature which would defeat the purpose of parallelizing annealing for better performance. However, the resulting temperature was empirically found to be constant over the circuit sizes tried; it corresponded to the 12th temperature in a 27 temperature

fixed temperature schedule of the simulated annealing approach that was tried out by them.

Another approach developed by the researchers to determine the starting temperature of annealing has been reported recently. The temperature of a given placement can be estimated by making use of the equilibrium characteristics of simulated annealing [51]. The approach involves starting at any temperature, and generating several hundered moves, and evaluating the total net cost change. If the net change is negative, then the temperature should be higher, and if positive, then the temperature should be lower. Using this approach, a binary search can be done to quickly converge to the correct starting temperature.

The final part of the parallel placement algorithm uses Section Annealing for the lower temperature region which begins with an interim placement. This placement is divided up geographically into subareas. The subareas and the cells contained in those areas are assigned to separate processors. The subareas were kept are square as possible. Each processor then generates simulated annealing style moves, in parallel, for the cells which it is assigned and tests those moves for acceptance. If a move is accepted, then the accepting processor transmits the move to other processors so that they can maintain a consistent view of the cell positions.

Natarajan and Kirkpatrick have proposed an interesting strategy for combining heuristics [52]. Using an implementation of a parallel simulated annealing algorithm based on the parallel algorithm of Darema *et al.* [32] discussed earlier which allowed acceptance of multiple moves that interact, they performed the following experiments on an 8 processor ACE workstation which is a shared memory multiprocessor supporting the Fetch-and-Add instruction in software. They performed a set of experiments on four placement problem instances having different connectivity characteristics and for whom the optimal solutions were known. This allowed the authors to compare the quality of the results produced by various heuristic approaches. Each of the instances were run a large number of times to study the statistical variation in the results. Some general conclusions were as follows.

In all instances of the placement problem, the quality of the placement solution deteriorated with increasing number of processors in a parallel simulated annealing run. In particular the average and standard deviation of the cost of the final solution tended to increase with the addition of processors. In all instances of the problem, the total computational effort measured by adding the amount of CPU seconds spent over all the participating processors per run of the algorithm showed a tendency to increase with increased number of processors. Since a decline in the quality of solution and increase in total work done per iteration are both undesirable characteristics of any parallel implementations of the simulated annealing algorithm, Natarajan and Kirkpatrick have studied an interesting extension of the algorithm.

They considered the use of multiple independent simulated annealing runs until the best solution is obtained meets a specified confidence level. Their general conclusion was the following. Given a P processor system, the best approach is to

divide the P processors into k groups each containing m processors ($k * m = P$). Run k independent and simultaneous runs of a simulated annealing algorithm and in each of these runs use m processors in parallel. Choose the best of the placements obtained in the k independent runs. The authors have determined an optimal value $m*$ for the parameter m that can be obtained experimentally. The authors studied an implementation on an 8 processor ACE workstation and reported that the optimal number for m is around 3.

7. Conclusions

In this paper we have reviewed some parallel algorithms for VLSI cell placement. Since the placement problem is *NP*-complete to begin with, numerous heuristic approaches have been proposed to solve it, while others are being continually developed. We have discussed some approaches to develop parallel algorithms for some of the better known heuristics for both shared memory and distributed memory multiprocessors.

We conclude this survey by making the following observations. While there are numerous heuristics for solving the placement problem, each heuristic has its own advantages. Methods based on constructive techniques such as min-cut and quadrisection are very fast, but they give poor quality of results. Methods based on probabilistic search techniques such as simulated annealing, simulated evolution or genetic approaches give very good quality of results, but take extremely long times. If the goal of the CAD designer is to produce layouts of the best quality, one has to use these extremely time consuming methods. That is exactly where parallel processing can help: to provide the best quality of results in the shortest runtimes.

A problem that is commonly faced in the design of efficient parallel algorithms for placement is that every year, better and better sequential heuristics are being discovered, which improve on previous methods in terms of quality of layouts produced, in terms of better objective functions, and in terms of runtimes. Unless the parallel algorithms can be designed such that these newer efficient techniques can be rapidly incorporated into the parallel implementations, the latter will become rapidly obsolete in terms of practical use.

Another problem with many of the parallel algorithms is that often they are targeted to execute on a specific parallel architecture for maximum performance, example a bus based shared memory multiprocessor, or a hypercube based distributed memory multiprocessor. As newer parallel machines are introduced with different interconnect topologies and parallel programming models, again there is a danger of the parallel algorithms becoming outdated.

Future work in the area of parallel placement algorithms therefore needs to address both the above mentioned problems. One solution to the first problem is to develop a framework for parallel placement algorithms where the basic research that needs to be performed is how to partition a placement problem into subproblems. The individual placement subproblems can be solved by the best possible sequential

methods as they become available. The interactions among the subproblems for various sequential heuristics is the important part of the parallel placement algorithm research.

A solution to the second problem is to develop parallel algorithms for an abstract parallel programming model that can be ported to a variety of parallel machines rapidly. Various efforts are underway in the parallel programming methodologies community to develop such abstract models.

Acknowledgements

The author wishes to express his sincere thanks to the following people who helped with the preparation of all the figures in this paper: Carolin Rouse, Ralph Kling, and Randall Brouwer.

References

1. B. T. Preas and P. G. Karger, "Automatic placement: A review of current techniques", *Proc. 23rd Design Automat. Conf.*, Jun. 1986, pp. 622–629.
2. M. Hanan and J. M. Kurtzberg, "Placement techniques", in *Design Automation of Digital Systems: Theory and Techniques*, ed. M. A. Breuer, Prentice-Hall, 1972, pp. 213–282.
3. M. R. Hartoog, "Analysis of placement procedures for VLSI standard cell layout", *Proc. 23rd Design Automat. Conf.*, Jun. 1986, pp. 314–319.
4. M. A. Breuer, "Min-cut placement", *Jour. Design Automation and Fault Tolerant Computing*, vol. 1, Oct. 1977, pp. 343–382.
5. B. W. Kernighan and S. Lin, "An efficient heuristic for partitioning graphs", *Bell Syst. Tech. Jour.*, vol. 49, Feb. 1970, pp. 291–307.
6. A. E. Dunlop and B. W. Kernighan, "A Procedure for placement of standard-cell VLSI circuits", *IEEE Trans. Computer-Aided Design of Circuits and Systems*, vol. CAD-4, Jan. 1985, pp. 92–98.
7. M. Hanan and J. M. Kurtzberg, "A review of the placement and the quadratic assignment problem", in *SIAM Review*, Apr. 1972, pp. 324–342.
8. D. G. Schweikert, "A 2-dimensional placement algorithm for the layout of electrical circuits", *Proc. 13th Design Automation Conf.*, Jun. 1976, pp. 408–416.
9. M. Hanan, P. K. Wolff Sr. and B. J. Agule, "Some experimental results on placement techniques", *Proc. 13th Design Automation Conf.*, Jun. 1976, pp. 214–224.
10. C. Cheng and E. Kuh, "Module placement based on resistive network optimization", *IEEE Trans. Computer-Aided Design*, vol. 3, Jul. 1984, pp. 218–225.
11. R. Tsay, E. Kuh and C. Hsu, "Module placement for large chips based on sparse linear equations", *Int. Jour. Circuit Theory Appl.*, vol. 16, 1988, pp. 411–423.
12. S. Kirkpatrick, C. D. Gelatt and M. P. Vecchi, "Optimization by simulated annealing", *Science*, vol. 220, May 1983, pp. 671–680.
13. M. D. Huang, F. Romeo and A. Sangiovanni-Vincentelli, "An efficient general cooling schedule for simulated annealing", *Proc. Int. Conf. Computer-Aided Design*, Nov. 1986, pp. 381–384.
14. C. Sechen and A. Sangiovanni-Vincentelli, "The TimberWolf placement and routing package", *J. of Solid-State Circuits*, vol. 20, 1985, pp. 510–522.
15. C. Sechen and A. Sangiovanni-Vincentelli, "TimberWolf3.2: A new standard cell placement and global routing package", *Proc. 23rd Design Automation Conf.*, Jun. 1986, pp. 432–439.

16. C. Sechen and K. W. Lee, "An improved simulated annealing algorithm for row-based placement", *Proc. Int. Conf. Computer-Aided Design*, Nov. 1987, pp. 471–481.
17. L. K. Grover, "A new simulated annealing algorithm for standard cell placement", *Proc. Int. Conf. on Computer-Aided Design*, Nov. 1986, pp. 378–380.
18. R. M. Kling and P. Banerjee, "ESP: A new standard cell placement package using simulated evolution", *Proc. 24th Design Automation Conf.*, Miami Beach, FL, Jun. 1987, pp. 60–66.
19. R. M. Kling and P. Banerjee, "Concurrent ESP: A placement algorithm for execution on distributed processors", *Proc. Int. Conf. on Computer-Aided Design*, Santa Clara, CA, Nov. 1987, pp. 354–357.
20. R. M. Kling and P. Banerjee, "ESP: Placement by simulated evolution", *IEEE Trans. Computer-Aided Design*, vol. 8, Mar. 1989, pp. 245–256.
21. R.M. Kling and P. Banerjee, "Optimization by simulated evolution with application to standard cell placement", in *27th Design Automation Conference*, Orlando, FL, Jun. 1990, pp. 20–25.
22. J. P. Cohoon and W. D. Paris, "Genetic placement", *IEEE Trans. Computer-Aided Design*, Nov. 1987, pp. 956–964.
23. K. Shahookar and P. Mazumder, "A genetic approach to standard cell placement using meta-genetic parameter optimization", *IEEE Trans. Computer-Aided Design*, vol. 9, May 1990, pp. 500–511.
24. K. Shahookar and P. Mazumder, "VLSI placement techniques", *ACM Computing Surveys*, vol. 23, Jun. 1991, pp. 143–220.
25. B. Preas and M. Lorenzetti, *Physical Design Automation of VLSI Systems*, Menlo Park, CA: Benjamin-Cummings Publishing Co., 1988.
26. T. C. Hu and E. S. Kuh, *VLSI Circuit Layout*, New York, NY: IEEE Press, 1985.
27. D. J. Chyan and M. A. Breuer, "A placement algorithm for array processors", *Proc. 20th Design Automation Conf.*, Jun. 1983, pp. 182–188.
28. K. Ueda, T. Komatsubara and T. Hosaka, "A parallel module placement approach for logic module placement", *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. CAD-2, Jan. 1983, pp. 39–47.
29. C. Sechen and A. Sangiovanni-Vincentelli, "The TimberWolf placement and routing package", *Proc. Custom Integrated Circuits Conf.*, May 1984, pp. 522–527.
30. S. A. Kravitz and R. A. Rutenbar, "Placement by simulated annealing on a multiprocessor", *IEEE Trans. Computer-Aided Design*, vol. CAD-6, Jun. 1987, pp. 534–549.
31. M. D. Durand, "Controlling error in parallel simulated annealing algorithms for VLSI placement", *IEEE Design and Test*, (to appear).
32. F. Darema, S. Kirkpatrick and V. A. Norton, "Parallel algorithms for chip placement by simulated annealing", *IBM Jour. Res. Dev.*, May 1987.
33. J. Sargent and P. Banerjee, "A parallel row-based algorithm for standard cell placement with integrated error control", in *Proc. 26th Design Automation Conf.*, Las Vegas, NV, Jun. 1989, pp. 590–594.
34. T. F. Chan and Y. Saad, "Multigrid algorithms on the hypercube multiprocessor", *IEEE Trans. Computers*, vol. C-35, Nov. 11, 1986, pp. 969–977.
35. K. Natarajan and S. Kirkpatrick, "Evaluation of parallel placement by simulated annealing: Part I — The Decomposition Approach", IBM Tech. Report RC 15246, Nov. 1989.
36. A. Casotto, F. Romeo and A. Sangiovanni-Vincentelli, "A parallel simulated annealing algorithm for the placement of macro-cells", *IEEE Trans. Computer-Aided Design*, Sep. 1987, pp. 838–847.

37. P. Banerjee and M. Jones, "A parallel simulated annealing for standard cell placement on a hypercube computer", *Proc. Int. Conf. Computer-Aided Design*, Nov. 1986, pp. 34–37.
38. M. Jones and P. Banerjee, "Performance of a parallel algorithm for standard cell placement on the intel hypercube" *Proc. 24th Design Automation Conf.*, Miami Beach, FL, Jun. 1987, pp. 807–813.
39. P. Banerjee, M. H. Jones and J. S. Sargent, "Parallel simulated annealing algorithms for standard cell placement on hypercube multiprocessors", *IEEE Trans. Parallel and Distributed Systems*, vol. 1, Jan. 1990, pp. 91–106.
40. J. S. Rose, W. M. Snelgrove and Z. G. Vranesic, "Parallel standard cell placement algorithms with quality equivalent to simulated annealing", *IEEE Trans. Computer-Aided Design*, Mar. 1988, pp. 387–396.
41. C. P. Wong and R. D. Fiebrich, "Simulated annealing-based circuit placement on the connection machine system", *Proc. Int. Conf. Computer Design*, Oct. 1987, pp. 78–82.
42. A. Casotto and A. Sangiovanni-Vincentelli, "Placement of standard cells using simulated annealing on the connection machine", *Proc. Int. Conf. Computer-Aided Design*, Nov. 1987, pp. 350–353.
43. N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller and E. Teller, "Equations of state calculations by fast computing machines", *Jour. Chem. Physics*, vol. 21, 1953, pp. 1087–1091.
44. J. P. Cohoon, S. U. Hegde, W. N. Martin and D. Richards, "Distributed genetic algorithms for the floorplan design problem", *IEEE Trans. Computer-Aided Design*, vol. 10, Apr. 1991.
45. P. Suaris and G. Kedem, "A quadrisection-based combined place and route scheme for standard cells", *IEEE Trans. Computer-Aided Design*, vol, 8, Mar. 1989, pp. 234–244.
46. P. Suaris and G. Kedem, "An algorithm for quadrisection and its application to standard cell placement", *IEEE Trans. Circuits and Systems*, vol. 35, Mar. 1988, pp. 294–303.
47. R. J. Brouwer and P. Banerjee, "PARAGRAPH: A parallel algorithm for simultaneous placement and routing using hierarchy", in *Proc. European Design Automation Conf. (EDAC-92)*, Brussels, Belgium, Mar. 1992.
48. M. Burstein and R. Pelavin, "Hierarchical wire routing", *IEEE Trans. Computer-Aided Design*, vol. CAD-2, no. 4, Oct. 1983, pp. 223–234.
49. R. J. Brouwer and P. Banerjee, "FIGURE: A parallel hierarchical global router", in *27th Design Automation Conference*, Orlando, FL, Jun. 1990, pp. 360–364.
50. C. P. Ravikumar and S. Sastry, "Parallel placement on hypercube architectures", in *Proc. Int. Conf. Parallel Processing (ICPP89)*, St. Charles, IL, Aug. 1989, pp. III:97–III:100.
51. J. Rose, W. Klebsch and J. Wolf, "Temperature measurement and equilibrium dynamics of simulated annealing as placements", *IEEE Trans. Computer-Aided Design*, vol. 9, Oct. 1990, pp. 253–259.
52. K. Natarajan and S. Kirkpatrick, "Evaluation of parallel placement by simulated annealing : Part II — The flat approach", IBM Tech. Report RC 15247, Nov. 1989.

Algorithmic Aspects of VLSI Layout
Eds. M. Sarrafzadeh and D. T. Lee
© World Scientific Publishing Co.

APPROXIMATE SOLUTIONS FOR GRAPH AND HYPERGRAPH PARTITIONING

FILLIA MAKEDON
Department of Mathematics and Computer Science
Dartmouth College
Hanover, NH 03755, USA

and

SPYROS TRAGOUDAS
Computer Science Department
Southern Illinois University at Carbondale
Carbondale, IL 62901, USA

ABSTRACT

Recently, Leighton and Rao introduced a multicommodity flow technique that initiated research activities towards the development of fast approximate solutions for balanced bipartitioning and multiway partitioning problems on graphs and circuits. Leighton and Rao's algorithm gives balanced bipartitions which are within polylogarithmic factors of the optimal only for graphs with uniform weighted nodes. Moreover, its time complexity is a high order degree polynomial of the input size. Naturally, the improvements on Leighton and Rao's algorithm can be categorized as: (a) Approximations for general graph bipartitioning, (b) approximations for hypergraph bipartitioning, (c) extension of (a), (b) to multiway partitioning and (d) faster algorithms for all the approximations in (a)–(c). The approaches for (a)–(c) follow a two phase generic: In Phase 1 we first define and solve a concurrent flow problem. The flow problem is then used in an algorithm that approximates the flux of the input graph or hypergraph. In Phase 2, we use the approximate flux to derive the final partitions. The time complexities of the above algorithms are determined by the time to solve the concurrent flow problem. Therefore research efforts in (d) focus on fast approximate solutions for the latter problem. The fastest algorithms have time complexity quadratic to the number of nodes, up to polylogarithmic factors.

In this paper, we initially focus on approximation algorithms for general graph bipartitioning and we describe an algorithm that obtains a polylogarithmic times optimal approximation for this problem. Then we survey the existing approximation algorithms in (a)–(c), including Leighton-Rao's approach. Finally, we describe recent research efforts for improving the time complexities of the approximation algorithms.

Keywords: Balanced bipartitioning, multiway partitioning, multicommodity flows.

1. Introduction

A *graph* $G = (V, E)$ consists of n nodes and m *edges*. Similarly, a *hypergraph* $H = (V, E_h)$ consists of n nodes and m *hyperedges*. A hyperedge e in E_h is defined

by a subset of nodes V_e in V. *Partitioning* is the task of dividing a given graph $G = (V, E)$ or hypergraph $H = (V, E_h)$ into smaller parts. The objective is to partition V into sets so that the sum of the weights on edges or hyperedges connecting nodes in different sets is minimized. This is the *cost* of the partition.

In VLSI we are operating on *electronic circuits.*[16] A circuit, more often given as a *netlist*, is a description of *switching elements* and their connecting *wires*. The notion of set size is formalized by assigning weights on the elements. The sum of the weights on the elements in the set is the *size* of the set. In *circuit partitioning* the objective is to partition the elements into sets such that the *sizes* of the sets are within prescribed ranges and the sum of the weights on the interconnecting wires is minimized. The weights on the elements may represent the area of the element.[25] Additionally, the weight on a net may express parameters such as the *buswidth* of the net, i.e., the number of bits that can be sent across the net in parallel. It may also be used to assign higher priorities to nets that should not run across different sets.

Circuit partitioning is abstracted and formalized as an operation on hypergraphs. An *exact* or an *approximation* algorithm for hypergraph partitioning is also an exact or an approximation algorithm for circuit partitioning. Thus, the algorithms described in this paper for hypergraph partitioning apply for circuit partitioning as well. However, there are several VLSI applications in which partitioning is performed on graphs rather than hypergraphs. For example, in routing we are interested in partitioning *routing graphs* where the nodes represent *switchboxes* for wire routing and the edges represent *routing channels.*[16]

In this paper, we describe partitioning algorithms on graphs and hypergraphs with integer weights on the nodes and the edges or hyperedges. In our partitioning problems we are always required to *partition into sets of sizes within predefined ranges.* The latter is central in *VLSI layout.* Partitioning is used to divide a graph or hypergraph *hierarchically* into sets using divide and conquer algorithms for placement, floorplanning, and other layout problems.[16] This divide and conquer scheme requires *balanced bipartitions* where the goal is to partition to two almost equal size sets. More formally, we partition into two sets for which the ratio of their sizes is bounded by a constant. We also consider the problem of partitioning into $p \geq 3$ sets, $V_1, \ldots, V_p$, so that the size of each set is less than or equal to an integer $k \geq 3$. This partitioning problem is called *multiway partitioning.* There are two versions of multiway partitioning. In Version 1, we want to minimize the sum $\sum_{V_i} w(e)$, where $w(e)$ is the weight on edge or hyperedge e external to V_i. In the second version, we minimize the sum $\sum w(e)$, where $w(e)$ is the weight of edge or hyperedge e external to at least one set V_i, $1 \leq i \leq p$. The two versions are equivalent if the input is a graph G. Applications of Version 1 are mentioned in Refs. 16, 25 and of Version 2 in Refs. 8, 25. We describe approximation algorithms for both versions of the problem. These approximation algorithms can also be applied for the special case of multiway partitioning when we partition the input graph or hypergraph into $p \geq 3$equal size sets. We call this problem the *equal size*

multiway partitioning problem. Applications of the latter problem are described in Ref. 16.

The multiway partitioning problem and the bipartitioning problem into two equal size sets have been shown to be NP-hard on general graphs and hypergraphs.[6] The case when the size of the sets does not exceed two can be solved polynomially using *matching*.[6] Thomas Lengauer describes in his book polynomial time algorithms for special cases of graphs.[16] Many *heuristic* solutions have also been proposed for *balanced bipartitioning* and (*equal size*) *multiway partitioning*.[11,4,2,1,9] Most of these heuristics have been presented for the balanced bipartitioning problem but they can be extended to handle multiway partitioning. These heuristic solutions do not have provably good performance but work well in practice.

Little has been done so far in the area of proposing *approximation algorithms* for graph and hypergraph partitioning. The aim of this paper is to describe a new framework that guarantees polylogarithmic times optimal approximations for graph and hypergraph balanced bipartitioning and multiway partitioning.[21,17,23,19,27] In this framework we use *multicommodity flows*. A multicommodity flow problem on a graph with capacities on its nodes and edges, consists of a set of *commodities* each defined by a triple (*source, destination, demand*). The goal is to route the commodities subject to the capacity constraint so that the demand of each commodity is satisfied. The graph or hypergraph in a partitioning instance, however, does not have any capacity constraints. The assignment of the capacities in the input instance will be done in a way to be described later and which guarantees the approximation bounds of our solutions.

In the partitioning framework, we use multicommodity flow techniques to approximate the *flux*. The flux is a partitioning related quantity that we define on graphs and hypergraphs which represents the *expansion* of the graph. We say that a graph $G = (V, E)$ has flux a if every set A of nodes with weight at most half of the total weight is connected to $V - A$ with edges of weight at least $a \cdot l(A)$, where $l(A)$ denotes the sum of the weights on the nodes in A. The flux is defined similarly on hypergraphs. The flux is also called the *minimum edge expansion* of the graph G. It is NP-hard to compute the flux optimally for general graphs and hypergraphs.[22] However, we show here how to obtain an $O(\log n \cdot \log l(V))$ approximation for the flux, where $l(V)$ is the total weight on the nodes. In Refs. 17, 19, 23, 27 the authors describe similar algorithms that reduce the approximation factor, they obtain an $O(\log(\max\{m, n\})$ approximation of the flux when the input is a hypergraph H of m edges and n nodes.

In the second phase of the described technique, we use the computed approximate flux to derive polylogarithmic times optimal approximation algorithms for balanced bipartitions and multiway partitions. Since the approximation bounds for the latter algorithms depend on the approximate solution obtained in Phase 1, it is essential to derive efficient flux approximations. This has been the center of many recent research papers.[21,23,19,27] All of them are based on multicommodity flows. In Table 1 we summarize the flux approximation algorithms developed so far.

To our knowledge, flux approximation is the only technique that achieves approximate solutions for partitioning graphs and hypergraphs into prescribed set sizes. Recently, Cheng and Wei[3] presented a heuristic that derives balanced bipartitions based on a quantity similar to the flux. They consider the *sparsest cut* which is the minimum among all possible bipartitions ratio: sum of the weights on the edges in the cut that forms the bipartition divided by the product of the sizes of the two sets in the bipartition. However, the authors do not approximate the latter quantity. In this paper we show that the multicommodity flow framework can be used to approximate both the flux and the sparsest cut. We define multicommodity flow problems for which the flux and the sparsest cut are within a constant factor of each other. Therefore the flux approximation algorithms presented in this paper obtain polylogarithmic approximations of the sparsest cut as well.

Table 1. Flux approximation algorithms.

Input instance	Approximation bound	Reference
Graphs with unit node weights	$O(\log n)$	21
Hypergraphs with unit nodes weights	$O(\log(\max\{n,m\}))$	23
General graphs	$O(\log W(\mathrm{e}) \cdot \log l(V))$	12
General graphs	$O(\log n \cdot \log l(V))$	this paper
General graphs	$O(\log n)$	19
General hypergraphs	$O(\log(\max\{n,m\})$	18

n = number of nodes, m = number of edges or hyperedges, $l(V)$ = total node weight
$W(\mathrm{e})$ = total edge weight

The efficiency of an approximation algorithm is measured in terms of the quality of the approximation bound, and its worst-case time complexity analysis. Recently, the time complexity of all the flux approximation algorithms[21,17,23] (listed in Table 1) has been improved and is now acceptable for VLSI layout applications. The computationally intensive part of the algorithms is a subroutine which approximates the *optimization version of a multicommodity flow problem* that we define in order to approximate the flux. In our multicommodity flow problem, we need to define the commodities, the demand of each commodity as well as the capacity constraints. Recall, however, that a multicommodity flow problem is a *decision problem* and we are only examining whether there exists a routing of the commodities so that the demands are satisfied.

The optimization version of a multicommodity flow problem is called the *concurrent flow* problem. In a concurrent flow problem, the goal is to route all commodities in such a way so that we maximize the minimum demand percentage. Let z be the minimum demand percentage satisfied in a routing of the commodities, and z^* denote the maximum percentage z in the concurrent flow problem. We aim to a

routing of the commodities so that z is maximized. We call z^* the *throughput* of the concurrent flow problem.

Consider an algorithm that finds a routing of the commodities which ships $(1-\epsilon)\cdot z^*$ percent of each demand, where $0 < \epsilon < 1$. Such an algorithm is called an $O(1-\epsilon)$ approximation multicommodity flow algorithm. In the algorithms described in this paper, we actually need to obtain only an $O(1-\epsilon)$ approximation multicommodity flow algorithm.

Let a multicommodity flow problem with K commodities on a graph G with n nodes and m edges. Linear programming can be used to solve the problem optimally in polynomial time. Kapoor and Vaidya[15] gave a method to speed up the matrix inversions involved in Karmakar type algorithms for multicommodity flow problems. Combining their techniques with Vaidya's new linear programming algorithm that uses fast matrix multiplication[29] yields a time bound of $\tilde{O}(K^{3.5} \cdot n^3 \cdot m^{.5})$ for the concurrent flow problem with integer demands and an $\tilde{O}\ (K^{2.5} \cdot n^2 \cdot m^{.5})$ time bound for the $O(1-\epsilon)$ approximation. In the latter case the polylogarithmic factor depends on ϵ^{-1}. In our multicommodity flow problem $K = O(n^2)$. Clearly, the above time bounds are impractical for the sizes of n and m in graphs and hypergraphs for VLSI applications. We can reduce K to n and improve the above mentioned time bounds. However, the flux that we obtain will only optimal with very high probability.[19] This commodity reduction is not sufficient still.

However, recent combinatorial approximations for the concurrent flow problem make the framework practical.[13,14,19] All these combinatorial techniques are based on the simple idea of starting with an initial routing of the commodities and then rerouting the commodities so that the minimum demand percentage z is successively increased.[26] In Table 2 we show the time complexities for the existing flux approximation algorithms whose approximation bounds have been listed in Table 1. The algorithms are depending on different subroutines used to approximate the concurrent flow problem. The results are listed in chronological order. All but the first polylogarithmic factors depend on ϵ^{-1}. We use LP, C to denote that the concurrent flow problem has been approximated using linear programming and rerouting techniques, respectively. The algorithms can be either randomized or deterministic. This is denoted in Table 2 by R or D, respectively. In order to compute the time complexities of the partitioning algorithms, we have to consider an additional $O(\log n)$ factor for the balanced bipartitioning algorithm and an additional $O(n \cdot \log n)$ factor for the multiway partitioning algorithms.

Similar results can be obtained when the input instance is a hypergraph. The hypergraph flux approximation problem requires the solution of a concurrent flow problem on a graph which is generated from the hypergraph as is indicated later on. The time complexities will be higher now since the generated graph has $O(m \cdot n)$ edges and $O(n+m)$ nodes.

Before we proceed with the description of the partitioning framework, we mention that Rao[24] has proposed better approximations for planar graph bipartitions. Tragoudas *et al.*[28] have also presented multiway graph and hypergraph partitioning

Table 2. The time complexities of the flux approximation algorithms for graphs.

Input instance	Flow routine	Flux algorithm	Time	R/D	LP/C
General	29	21,19,12,27	$\tilde{O}(n^{6.5} \cdot m^{.5})$	D	LP
Unweighted	26	Special case of[21]	$O(n \cdot m^7 \cdot \epsilon^{-5})$	D	C
Unweighted	14	Special case of[21]	$\tilde{O}(n^3 \cdot m)$	D	C
Unweighted	14	Special case of[21]	$\tilde{O}(m^2)$	R	C
General	19	19,27,12	$\tilde{O}(n^3 \cdot m)$	D	C
General	19	19,27,12	$\tilde{O}(n^2 \cdot m)$	R	C
Unweighted nodes	19	21	$\tilde{O}(n \cdot m \cdot d)$	R	C
Unweighted	19	special case of[21]	$\tilde{O}(n^2 \cdot m^{1.5})$	D	C
Unweighted	19	special case of[21]	$\tilde{O}(n \cdot m^{1.5})$	R	C

n = number of nodes, m = number of edges, d = max node degree

approximations into sets of size at most a constant k which are within a constant factor from the optimal. In multiway hypergraph partitioning, the approximation bounds hold for the second version only. Finally, Leighton *et al.*[17] give approximation algorithms for planar graph and hypergraph multiway partitioning based on node deletion arguments.

This paper is organized as follows: In Sec. 2, we focus on the flux. In Sec. 2.1, we define the flux on graphs formally and then we show how flux approximations can lead to approximate balanced graph bipartitions. In Sec. 2.2 we describe the connection of the flux to a multicommodity flow problem which we define on the input graph. The latter is essential for the flux approximation. In Sec. 3, we present a flux approximation algorithm for general graphs in detail. We also show how the general graphs algorithm uses an approximation for the concurrent flow problem which was defined on the input graph. In Sec. 4, we survey other flux approximation algorithms. We define the flux on hypergraphs and we extend the flux approximation results to hypergraphs. In Sec. 5, we discuss multiway partitioning algorithms. In Sec. 6, we describe combinatorial techniques for approximating the concurrent flow problem fast. We conclude in Sec. 7.

2. Flux: A Basic Quantity for the Framework

In this section, we define the flux (or minimum edge expansion) a on graphs and hypergraphs. This is a partitioning related quantity and has already been shown by Rao[24] that approximate solutions for the flux can serve as the basis of a simple divide and conquer approach that guarantees approximate solutions for balanced bipartitioning. In Sec. 2.1 we briefly present Rao's technique. Rao's technique can also be applied to hypergraphs provided that we are able to approximate the flux on hypergraphs. In Sec. 2.2 we define a multicommodity flow problem on the input

graph as well as the corresponding *concurrent flow* problem. The concurrent flow problem must be defined in a way that we are able to obtain an approximation to the flux a. Clearly, the flux is a quantity that is not related to multicommodity flows. It is however related to the sparsest cut of the graph as we describe in Sec. 2.2. We show that an appropriate definition of the commodities in the flow problem suffices to unify the sparsest cut with the *minimum cut ratio* S, a partitioning quantity defined in the context of the multicommodity flow problem which we are able to approximate efficiently. To approximate the minimum cut ratio S, we must, at a preprocessing step, approximate the concurrent flow problem within an $O(1-\epsilon)$ factor, $0 < \epsilon < 1$. This way we succeed to approximating the flux.

2.1. The Flux and Balanced Bipartitions

Let $G = (V, E)$ be a graph with weight $l(v)$ on any node $v \in V$, $l(V)$ total weight on the nodes and weight $w(e)$ on any edge $e \in E$. We say that $G = (V, E)$ has flux a if every set of nodes $A \subset V$ such that $l(A) \leq l(V)/2$ is connected to $V - A$ with edges of weight at least $a \cdot l(A)$, where $l(A)$ denotes the sum of the weights on the nodes of A. We use $\Gamma(a)$ to denote a cut which corresponds to the flux a. This cut consists of the edges with endpoints in both sets of the bipartition that defines the flux a. From the above definition we have that the flux a is a bipartitioning related quantity which describes the expansion of G.

Every set V_1 of nodes in V defines a cut, denoted by $\Gamma(V_1)$, which is the set of edges with exactly one endpoint in V_1. Let $w(\Gamma(V_1))$ denote the sum of the weights on the edges with exactly one endpoint in V_1. For each cut $\Gamma(V_1)$ there is an *edge expansion* $a_{V_1} = w(\Gamma(V_1))/\min\{V_1, V - V_1\}$.

A more formal definition of the flux a is as follows: Let $V_1 \subseteq V$, $l(V_1)$ be the size of V_1 and $w(e)$ denote the weight on any edge e that connects V_1 with $V - V_1$.

$$a = \min_{V_1 \subset V} \{a_{V_1}\} = \min_{V_1 \subset V} \left\{ \frac{\sum^{e \in \Gamma(V_1)} w(e)}{\min\{l(V_1), l(V - V_1)\}} \right\} \tag{1}$$

Thus the flux is the minimum edge expansion of the graph G.

Consider now the balanced graph bipartitioning problem on G and let β, $l(V)$ be an integer β greater than or equal to three and the sum of the weights on the nodes of G, respectively. In *β-balanced bipartitioning* ($\beta - BB$) the size of each set must be greater than or equal to $\lfloor l(V)/\beta \rfloor$. Let α denote a constant greater than one. We similarly use $\alpha\beta - BB$ to denote a bipartitioning in which neither of the two sets in the partition has size less than $\lfloor l(V)/(\alpha \cdot \beta) \rfloor$. The algorithm presented in this section, is due to Rao.[24] It obtains an $O(f)$ times optimal approximation for the $\alpha\beta - BB$ problem, provided that we have an $O(f)$ approximation to the flux a, where f is a polylogarithmic function of the input size. We call this algorithm the Balanced Bipartitioning (BB) algorithm and, although it is described here for graphs, it also works on hypergraphs.[27] Algorithm BB maintains a set $\mathcal{T}$, initially empty.

Algorithm Balanced Bipartitioning (BB)

Step 1. Obtain an $O(f)$ approximation to the flux a of graph G. Let a_{appr} be the approximate flux.

Step 2. Consider the induced subgraphs $G_1 = (V_1, E_1)$ and $G_2 = (V - V_1, E_2)$, where $\Gamma(V_1) = \Gamma(V_2) = \Gamma(a_{appr})$. Let us assume that $l(V_1) \geq l(V_2)$. Set $\mathcal{T} = \mathcal{T} \cup V_2$. If the sizes of V_1 and $\mathcal{T}$ allow us to claim an $\alpha\beta - BB$, then exit algorithm BB; The balanced bipartition is defined by the sets V_1 and $\mathcal{T}$. Otherwise, set $G = G_1$ and go to Step 1.

End of algorithm Balanced Bipartitioning (BB)

Rao[24] showed that algorithm BB always obtains a $\alpha\beta - BB$ for predefined integer values of α and β. Then he showed the following theorem.

Theorem 1.[23,18,24] *Algorithm BB achieves an $\alpha\beta$-balanced bipartition which is within a $\Theta(f)$ factor to the optimal β-balanced bipartition.*

We briefly sketch the proof of Theorem 1. A more detailed proof can be found in Ref. 24.

Let OPT_β be the optimal β-balanced bipartition of G corresponding to a cut $\Gamma(V_1)$ and $|\text{OPT}_\beta|$ be its cost. Assume that at the i^{th} recursive step we find a partition with cost $\theta_i \cdot |\text{OPT}_\beta|$. Let N_i be the total weight on the nodes that are still needed for the balanced partition, i.e., N_i is the sum of the weights on the nodes that need to be removed from the node set on which we are currently iterating in order to achieve the $\alpha\beta$-balanced bipartitioning. Let $l(V)$ be the total weight on the nodes of G and l_i the total weight on the nodes that have been removed up to this step. Assume that algorithm BB terminates after I recursive calls. We use now a lemma shown by Rao in Ref. 24.

Lemma 1.[24] *Assuming that we were able to compute the flux a of G optimally, any iteration of the BB algorithm creates a bipartition which costs at most $|\text{OPT}_\beta|$, where β is an integer ≥ 3. Furthermore, both $l(V_1)$ and $l(V - V_1)$ are in the range $[N_i,\ l(V) - l_i - N_i]$.*

Assume initially that at each recursive call of algorithm BB we are able to find the flux a. We will use Lemma 1 to show that we can obtain an approximation of the flux a within an $(1 + \log(\alpha))$ factor, where α is a small constant. The argument is as follows: If Algorithm BB does not terminate on some recursive call, then we get θ_i of the total weight on the nodes still needed to be removed from the node set that we are currently partitioning. (The cut $\Gamma(a)$ separates at least as much weight proportional to each cost as the optimal balanced bipartitioning $|\text{OPT}_\beta|$.) Thus, the total node-weight that needs to be removed after i iterations is expressed by

the following recurrence relation:

$$N_0 = \left\lfloor \frac{l(V)}{\beta} \right\rfloor, N_i = N_{i-1} - \theta_i \cdot N_{i-1} \ . \tag{2}$$

Equivalently, N_i can be written as

$$N_i = \left\lfloor \frac{l(V)}{\beta} \right\rfloor \cdot \prod_{j \le i} (1 - \theta_j) \ . \tag{3}$$

Let C be the cost of the partition found just before the final iteration. We also have that $C = (\sum_{i<I} \theta_j) \cdot |\mathrm{OPT}_\beta|$.

Now we can bound the cost of the $\alpha\beta$-balanced bipartition given by the BB algorithm. (We still assume that the flux a can be computed optimally.) We bound C with the condition $N_{I-1} > \lfloor l(V) \cdot (\alpha - 1)/(\alpha \cdot \beta) \rfloor$, since if we needed any less we would have an $\alpha\beta$-balanced bipartition. The last iteration may cost an extra $|\mathrm{OPT}_\beta|$. Thus, the real upper bound is $C + |\mathrm{OPT}_\beta|$. With algebraic manipulations (for details see Ref. 24) C can be shown to be less than $\log(\alpha/(\alpha - 1)) \cdot |\mathrm{OPT}_\beta|$. Therefore, the BB algorithm has total cost less than $(1 + log(\alpha/(\alpha - 1))) \cdot |\mathrm{OPT}_\beta|$. Equivalently, the solution has cost less than $c \cdot |B_m|$, for a constant $c = 1 + f \cdot (\alpha/(\alpha - 1))$.

All of the above computations were based on the assumption that we were able to find the flux optimally. However, we can only obtain an $O(f)$ approximation of the flux.

2.2. Multicommodity Flow Problems and the Minimum Cut Ratio

As mentioned in Sec. 1, a multicommodity flow problem $\mathcal{M}$ on a graph $G = (V, E)$, with capacity $c(e)$ on every edge $e \in E$ and capacity $c(v)$ on every node $v \in V$, consists of a set of *commodities*. We need to route the commodities so that we satisfy their demands subject to the capacity constraints on the edges. In the corresponding concurrent flow problem $\mathcal{C}$, the goal is to route all commodities so that we maximize the minimum demand percentage satisfied. Let z be the demand percentage satisfied in a routing of the commodities in $\mathcal{C}$ and z^* be the throughput of $\mathcal{C}$. In this paper, we assume that $\mathcal{C}$ and $\mathcal{M}$ are defined on a graph G with integer capacity constraints and no capacity constraints on the nodes.

Clearly, both $\mathcal{M}$ and $\mathcal{C}$ are problems that are not related to graph partitioning. However, given a flow problem $\mathcal{M}$ on a graph G, we define a bipartitioning related quantity which we call it the *minimum cut ratio* $\mathcal{S}$. The minimum cut ratio S, is a quantity that represents the minimum cut in any multicommodity flow problem on G. In $\mathcal{M}$, there are two quantities associated with any cut $\Gamma(V_1)$ defined by the set $V_1 \subset V$: (a). The capacity of the cut, denoted by $c(\Gamma(V_1))$, equal to the sum of the capacities on the edges that define the cut $\Gamma(V_1)$, and (b) the demand across the cut, denoted by $d(\Gamma(V_1))$. The minimum cut ratio S is the minimum among all

possible ratios capacity/demand which are associated with all possible cuts in G. More formally, let $c(e)$ denote the capacity of each edge e in E. The minimum cut ratio S is defined as:

$$S = \min_{V_1 \subset V} \left\{ \frac{\sum^{e \in \Gamma(V_1)} c(e)}{d(\Gamma(V_1))} \right\} = \min_{V_1 \subset V} \left\{ \frac{c(\Gamma(V_1))}{d(\Gamma(V_1)} \right\} . \tag{4}$$

We use $\Gamma(S)$ to denote the cut that obtains S. A preprocessing step of the proposed multicommodity flow technique is to define an adequate multicommodity flow problem $\mathcal{M}$ on a graph G so that the minimum cut ratio S and the flux a can be related. We actually want them to be within a constant factor of each other. This way, by following a procedure described later on, we are able to claim logarithmic approximations for the flux a.

We are now ready to define the multicommodity flow problem on G for which S and a are within a constant factor of each other. We call this flow problem the *Graph Multicommodity* ($\mathcal{GM}$) flow problem. In $\mathcal{GM}$ the capacities are defined as follows: The capacity $c(e)$ on every edge e equals to the weight $w(e)$ of e. There are no capacity constraints for the nodes of G. The demands and commodities are given below: For each ordered pair of nodes $(v,\ w)$, we define a commodity with demand $l(v) \cdot l(w)/2$, where $l(v)$ denotes the weight on node v in V. We call the optimization version of the $\mathcal{GM}$ flow problem the *Graph Concurrent* ($\mathcal{GC}$) flow problem. (The goal is to maximize the demand percentage that is satisfied.)

We now show that in $\mathcal{GM}$ the minimum cut ratio S and the flux a are within a constant factor of each other. Let $w(\Gamma(V_1))$ denote the sum of the weights on the edges in the cut $\Gamma(V_1)$. In $\mathcal{GM}$, the minimum cut ratio S can be rewritten as:

$$S = \min_{V_1 \subset V} \left\{ \frac{w(\Gamma(V_1))}{d(\Gamma(V_1))} \right\} = \min_{V_1 \subset V} \left\{ \frac{\sum^{e \in \Gamma(V_1)} w(e)}{l(V_1) \cdot l(V - V_1)} \right\} . \tag{5}$$

The last equality is derived since in $\mathcal{GM}$ the demand across $\Gamma(V_1)$ is $l(V_1) \cdot l(V - V_1)$. From Eqs. 1 and 5 we have that $a = S \cdot l(V - V_1)$ when $l(V_1) \leq l(V - V_1)$. Equivalently, by taking the maximum and minimum values for $l(V - V_1)$, we have that

$$\frac{l(V) \cdot S}{2} \leq a \leq l(V) \cdot S \tag{6}$$

This implies that the flux a and the minimum cut ratio S are within a constant factor of each other.

The main goal of the multicommodity flow technique described in this paper is to derive an approximation of the flux a through S. This is demonstrated through the algorithm of the following section for graphs with arbitrary integer weights on the nodes and on the edges. In general, all these algorithms have a procedure in common which computes sufficiently "short" routes for the commodities between all possible pairs of nodes on G. The distance of these routes is subject to a length

function $l(e)$ on the edges which is determined in more detail in Sec. 6. However, the computation of this length function involves a $O(1-\epsilon)$ approximation for the $\mathcal{GC}$ flow problem, where $0 < \epsilon < 1$, i.e., a feasible flow that ships at least $(1-\epsilon)\cdot z^*$ percent of each demand, where z^* is the maximum percentage obtainable. The connection of the latter problem to our initial flux approximation problem is discussed in the next section. However, we want to mention in advance that the $(1-\epsilon)$ approximation for the $\mathcal{GC}$ problem is critical since it determines the overall complexity of the multicommodity flow approach. Therefore, it is critical to obtain fast approximations for the $\mathcal{GC}$ problem. Similarly, the algorithms described in Secs. 4 and 5 involve $O(1-\epsilon)$ approximations for similar concurrent flow problems. For simplicity of presentation, we postpone until Sec. 6 the details of fast $O(1-\epsilon)$ approximations for the concurrent flow problems described in this paper.

3. A Flux Approximation Algorithm for General Graphs

Let a graph $G = (V, E)$ of n nodes, a weight $l(v)$ for every node $v \in V$, a weight $w(e)$ for every edge $e \in E$, and $l(V)$ be the sum of the weights on the nodes. The algorithm of this section obtains an approximate edge expansion which is within a $O(\log n \cdot \log l(V))$ factor of the flux. Consider the $\mathcal{GM}$ problem on G with capacities $c(e) = w(e)$ on the edges e in E. Let A be any subset of nodes of G. $\Gamma(A)$ denotes the cut defined by the set A, $c(\Gamma(A))$ denotes the capacity of the cut $\Gamma(A))$ and $d(\Gamma(A))$ is the sum of the commodity demands across the cut $\Gamma(A)$. Each commodity i in the $\mathcal{GM}$ problem is represented by the triple $\{(s_i, t_i, d(i)) : i = 1, \ldots, k\}$, where s_i, t_i and $d(i) = d(s_i, t_i)$ denote the source, sink and demand of commodity i, respectively. In $\mathcal{GM}$ we have that $d(s_i, t_i) = l(s_i) \cdot l(t_i)/2$. Let z^* be the throughput in $\mathcal{GM}$. The analog of the minimum cut in the corresponding $\mathcal{GC}$ is the set $A \subset V$ obtaining the minimum cut ratio $S = \min_A\{c(\Gamma(A))/d(\Gamma(A))\}$. We can observe that the value of z^* is at most S. In other words, if z^* is less than one, then the $\mathcal{GM}$ flow problem is infeasible.

We will use the above fact together with Eq. 6 and duality arguments for multicommodity flows to derive an approximation algorithm for the flux a on G. Think of each pair (s_i, t_i) of commodity endpoints as a *demand edge* $e_{(s_i,t_i)}$, with a weight $d(s_i, t_i)$ equal to the demand associated with the commodity. Let E_d be the set of the demand edges $e_{(s_i,t_i)}$. Let ℓ, $\text{dist}_\ell(u, v)$ be any length function assigning lengths to the capacity edges of G and the shortest path distance between nodes u and v subject to the function ℓ, respectively. We now state standard linear programming arguments for multicommodity flow problems as have been given by Iri in Ref. 10. Let z be the demand percentage satisfied in a routing of the commodities in $\mathcal{GM}$. The linear programming dual to the problem of maximizing z is minimizing the sum

$$\frac{1}{\sum^{e(s,t)\in E_d} \text{dist}_\ell(s,t) \cdot d(s,t)}, \tag{7}$$

subject to the constraint

$$\sum^{e} c(e) \cdot \ell(e) = 1 \ . \tag{8}$$

In particular, the value of Eq. 7 is always at least the value of z^*. When ℓ minimizes the sum in 7 subject to Eq. 8, and the flow routing maximizes z, then the sum in 7 equals to the thoughput z^*. That is, at *optimality* we have:

$$\frac{1}{\sum^{e(s,t)\in E_d} \text{dist}_\ell(s,t) \cdot d(s,t)} = z^* \leq S \ . \tag{9}$$

Equivalently, Eq. 9 can be written:

$$\frac{1}{\sum^{i} \text{dist}_\ell(s_i,t_i) \cdot d(s_i,t_i)} = z^* \leq S \ . \tag{10}$$

By Eqs. 10 and 6 we have that for the $\mathcal{GC}$ flow problem and at optimality

$$\frac{1}{\sum^{i} \text{dist}_\ell(s_i,t_i) \cdot d(s_i,t_i)} = z^* \leq \frac{2a}{l(V)} \ . \tag{11}$$

By Eq. 11, if we were able to find a set A with the edge expansion $a_A = c(\Gamma(A))/\min\{A, V-A\}$ corresponding to cut $\Gamma(A)$ so that the sum $\sum^{i} \text{dist}_\ell(s_i,t_i) \cdot d(s_i,t_i)$ is within a polylogarithmic factor of $l(V)/a_A$, then we have that a_A is a polylogarithmic times optimal approximation for the flux a. Let C, D, n, $\text{dist}_\ell(s_i,t_i)$, a_A denote the sum of capacities, the sum of the demands, the number of nodes, the shortest path between nodes s_i and t_i under the length function ℓ and the edge expansion a_A corresponding to cut $\Gamma(A)$, respectively. We present a polynomial time algorithm that finds a node set A such that

$$\sum^{i} \text{dist}_\ell(s_i,t_i) \cdot d(s_i,t_i) = O(\log n \cdot \log D) \cdot \frac{l(V)}{a_A} = O(\log n \cdot \log l(V)) \cdot \frac{l(V)}{a_A} \ . \tag{12}$$

The last equality in Eq. 12 holds since in $\mathcal{GM}$ we have that D is $O(l(V)^2)$. From Eqs. 10 and 12 we have that the cut $\Gamma(a_A)$ guarantees an approximate flux.

We give now the polynomial time algorithm that satisfies Eq. 12. The algorithm assumes a length function ℓ and an optimal flow computation to the $\mathcal{GC}$ problem so that Eq. 9 is satisfied. It is, however, time consuming to compute z^* optimally. (See Sec. 6 for more details and on how the optimal flow computation can determine a length function ℓ.)

Alternatively, we can work on a more relaxed version where we only need an $O(1-\epsilon)$ approximation for z^* in $\mathcal{GM}$. For any multicommodity flow problem, it has been proved[19] that if a routing of the commodities satisfies a demand percentage z, then we have that $z \cdot (\sum_{e\in E} \ell(e) \cdot c(e)) \geq \sum_{i=1}^{k} \text{dist}_\ell(s_i,t_i) \cdot d(s_i,t_i)$. We can therefore obtain a more relaxed problem formulation where we are still seeking for

an algorithm that obtains Eq. 12 but the routing of the commodities is such that the length function ℓ on the edges satisfies that

$$\frac{\sum^i d(s_i,t_i)\cdot \mathrm{dist}_\ell(s_i,t_i)}{\sum_{e\in E}\ell(e)c(e)} = \sum_i d(s_i,t_i)\cdot \mathrm{dist}_\ell(s_i,t_i) \geq \frac{1}{z^*\cdot(1+\epsilon)}\,, \qquad (13)$$

for a constant $0 < \epsilon < 1$. (The equality between the two leftmost terms is due to Eq. 8.)

In Refs. 19, 14, 13 it has been shown that for any concurrent flow problem it holds that if we find a routing of the commodities and a length function ℓ such that Eq. 13 is satisfied, then the demand percentage z satisfied is within an $O(1-\epsilon)$ factor of z^*. Therefore, in our relaxed problem formulation we are only looking for $a(1-\epsilon)$ times optimal approximation for z^* in $\mathcal{GM}$. The latter will also determine the lengths $\ell(e)$ on every edge e for which the algorithm described below achieves Eq. 12. (In Sec. 6 we give algorithms which compute the approximate flow and the length function ℓ.)

In what follows, we give an algorithm that satisfies Eq. 12. Let ℓ, be a length function that satisfies Eq. 13 and length $(P(s_i,t_i))$ be the length of a path $P(s_i,t_i)$ from node s_i to node t_i. We initially give an algorithm that assigns a path $P(s_i,t_i)$ to each commodity i such that $\sum^i \mathrm{length}(P(s_i,t_i))\cdot d(s_i,t_i) = O((1/S)\cdot\log n\cdot\log D) = O((1/S)\cdot\log n\cdot\log l(V)) =$ (by Eq. 6) $= O(\log n\cdot\log l(V)\cdot(l(V)/a))$. Then we will show how to derive an approximation of the flux a.

Our algorithm consists of stages. At each stage t, we have a concurrent flow problem with minimum cut ratio $S_{t+1} \geq S_t$. Let D_t be the sum of the demands of the commodities that comprise the flow problem at stage t. Initially, $D_t = D$. During the first stage, the concurrent flow problem is the input problem with minimum cut ratio S. At each stage we partition G into trees of depth $O(\log n/(S_t\cdot D_t)) \leq O(\log(n/(S\cdot D_t))$, and we route every commodity with both endpoints in the same tree on the unique path between the two nodes in the tree. The goal of a stage is to route at least half of the sum of the demands of the commodities that still remain unrouted at the beginning of the stage. The remaining demands define the concurrent flow problem for the next stage. This way we have at most $O(\log D)$ stages. We can think that for each pair of commodities (s_i,t_i) there exists a demand edge between nodes (s_i,t_i) of weight $d(s_i,t_i)$. We use variable t, initially set to zero, to distinguish between different stages of the algorithm. We stop initiating a new stage when no commodities are left to be routed. At the beginning of each stage, we split every edge into small pieces by assigning tokens and a capacity to each added token. This idea was introduced by Leighton and Rao Ref. 20 and is used to control the breadth first search expansion. Each stage corresponds of an iteration of steps $(a)-(e)$ in the algorithm below.

Let S_t be the minimum cut ratio at stage t. For the time being, we assume that we are able to compute S_t. Observe, however, that some capacities can be reduced without affecting the computation of S_t. For each node v at stage t compute

$r_v = \sum_w c(vw)/\sum_w d(vw)$ and $r^t = \min_v\{r_v\}$. Observe that if an edge e has capacity $c(e)$ more than $D_t \cdot r^t$, then it cannot participate in the minimum ratio cut S_t (it does not even participate in the computation of r^t) and its capacity can be reduced to $D_t \cdot r^t + 1$ without affecting the computation of S_t. We will take advantage of such capacity reductions. Note that the capacities will be resumed to their initial values when the next stage begins. The algorithm is as follows:

The routing algorithm

(a) Let $r = \min_v\{\sum_w c(vw)/\sum_w d(vw)\}$. For all edges e with initial capacity $c(e) > D_t \cdot r$, reduce their capacities to $D_t \cdot r + 1$. For notational convenience let C denote the sum of the capacities on the edges after the capacity reductions. Split each edge e of the input graph $G = (V, E)$ into discrete pieces by placing $\max\{1, \lceil \ell(e) \cdot C \rceil\}$ tokens. Therefore each piece has size C^{-1}. Also assign a capacity $c(e)$ on every token which lies on edge e.

(b) Select a node $v \in V$ such that $\sum^{w \in V} d(v, w)$ is maximum to initiate a set of nodes of G. If there is no node left go to Step (f).

(c) Extend the set by a level of tokens, expanding in a breadth first search manner. Thus, we expand on a breadth first search tree.

(d) Let i be the depth of the set generated that way, measured in tokens. Let C_i denote the token capacity of the set at level i. If $C_i > (1+((S_t \cdot D_t)/(4 \cdot C))$ C_{i-1}, then go to Step (c).

(e) Discard the nodes in the set. If the sum of the demands of the commodities with both endpoints in a set formed at Steps (b)–(d) is less than $D_t/2$ then go to Step (b).

(f) For each commodity with both endpoints in the same set, assign to the commodity the path in the set, and discard the commodity. Let $t = t+1$, and D_{t+1} be the sum of the demands of the remaining commodities. This set of demands will define a flow problem with an S_{t+1} value for stage $t + 1$. If $D_{t+1} \neq 0$ then go to Step (a) to initiate a new stage; else exit.

End of the routing algorithm

Observe that the routing algorithm, as described above, is no polynomial since the number of tokens per edge can be exponential to the input size. However, we can easily simulate the routing algorithm so that it runs in polynomial time. Observe also that if no capacity reduction has occurred at the beginning of a stage, then the term $\sum^{e \in E} c(e) \cdot \ell(e) = 1$. If some capacities are reduced, then the term $\sum^{e \in E} c(e) \cdot \ell(e) < 1$. Either way, the total capacity assigned to the tokens at some stage t is at most $\sum^{e \in E} c(e) \cdot (1 + \ell(e) \cdot C) \leq 2 \cdot C$. We also have that $(S_t \cdot D_t)/C < 1$ since a cut ratio can be obtained by considering the cut that includes all edges. Furthermore, the minimum cut ratio increases between successive stages since the demands decrease. A critical observation in our proof is that each time the algorithm starts expanding on a new set by picking a node v as a root of the

breadth first search tree at Step (b), we have that at least half of the sum of the demands D_t is not routed in the trees in the generated sets at stage t.

Next, we state a lemma similar to one proven in Ref. 12 which shows that after the algorithm partitions G in a collection of sets T (now stage t ended because there were no more nodes left), the sum of the demands routed in the sets will be at least $D_t/2$.

Lemma 2. *If during stage t the routing algorithm partitions G into sets T, then $D_{t+1} \leq D_t/2$.*

Proof. Let E_T, $c(E_T)$, $c(T)$ denote the set of edges incident to a set T discarded at stage t, the sum of the capacities on the edges in E_T, the total token weight in T, respectively. From Step (d) we have that $c(E_T) \leq (S_t \cdot D_t/(4 \cdot C)) \cdot c(T)$. Let Γ_t be the cut that corresponds to the partition into sets at the end of stage t and let $c(\Gamma_t)$ be the total capacity on the edges in the cut $c(\Gamma_t)$. If the sets T were edge-disjoint, then $c(\Gamma_t) = \sum_T c(E_T) \leq (\sum_T S_t \cdot D_t/(4 \cdot C)) \cdot \sum_T c(T) \leq S_t \cdot D_t/2$, where the last inequality occurs since the total token weight cannot exceed $2 \cdot C$. Observe that this is the worst case scenario for an upper bound for $c(\Gamma_t)$ since if an edge was in two E_T (it cannot be in more than two), then the capacity of the edge only needs to be encountered once in our upper bound computation. By definition of S_t, we have that $S_t \cdot D_{t+1} \leq c(\Gamma_t) \leq S_t \cdot D_t/2$. (The last part of the inequality is due to the previous inequality chain.) Therefore, we can conclude that $D_{t+1} \leq D_t$. □

We use *ln* to denote the natural logarithm log_e. We use Lemmas 2 and the previously mentioned observations to obtain some lemmas. Let c_T denote the sum of the capacities on the edges incident to the root node v, the root of a tree T generated during some stage t. ($c(e)$, C and c_T represent reduced capacities.)

Lemma 3. *At each stage t, $\sum_{e \in E} c(e) = C < n^3 \cdot c_T$.*

Proof. For each node v at stage t, let $r_v = \sum_w c(v,w)/\sum_w d(v,w)$ and $r^t = \min_v\{r_v\}$. After the capacity reductions at stage t we can assume that $C < m \cdot (D_t \cdot r^t + 1) < n^2 \cdot (D_t \cdot r^t + 1)/2$, where m, n is the number of edges and nodes of G, respectively. Thus for the selected node v that initiates the creation of a new set during the stage, we have that $C < n^2 \cdot (D_t \cdot r_v + 1)/2 = n^2 \cdot (D_t \cdot c_T/(\sum_w d(v,w)) + 1)/2$. The moment we are choosing node v, we have that at least $D_t/2$ demands are in the induced graph on which we are expanding. Let D', n' denote the sum of the demands and the number of nodes on the graph on which we are expanding. We have that $\sum_w d(v,w) \geq 2 \cdot D'/n' \geq D_t/n' \geq D_t/n$. (The leftmost pair of the inequality chain holds by the selection of node v.) The above two inequality chains are combined to give that $C < n^3 \cdot (c_T + 1)/2 < n^3 \cdot c_T$. □

Lemma 4. *We have that $\sum_i \text{length}(P(s_i,t_i)) \cdot d(s_i,t_i) \leq 64 \cdot \log D \cdot \ln n/S$.*

Proof. By Lemma 2, we only need to show that the length of every path assigned at stage t is at most $64 \cdot \ln n/(D_t \cdot S_t)$. Since the sum of the demands of the commodities i that were assigned at stage t is $D_t - D_{t+1}$, the above upper bound on the path length guarantees that $\sum_i \text{length}(P(s_i,t_i)) \cdot d(s_i,t_i) \leq 64 \cdot \ln n/S_t \leq 64 \cdot \ln n/S$. The last inequality holds since $S_t \geq S$, $\forall t \geq 1$. We prove the upper bound on the path length by showing that depth d of the breadth first tree generated at stage t is at $32 \cdot \ln n/(D_t \cdot S_t))$. If we express the depth d in number of tokens, we need to show that the depth d is $O(C \cdot \ln n/(D_t \cdot S_t)$.

We prove the latter by a contradiction argument. Assume that the depth d of such a tree is $O(C \cdot \ln n)/(S_t \cdot D_t))$. If we have not terminated at depth d, then from Step (d) of the algorithm, we have that $C_d > (1 + (S_t \cdot D_t/(4 \cdot C)))^{d-1} \cdot c_T$, where C_d is the sum of the capacities of the tokens in the tree generated during the stage and up to depth d.

But we show that the quantity $(1 + (S_t \cdot D_t/(4 \cdot C)))^{d-1} \cdot c_T$ is greater than $2 \cdot C$, the total capacity of the tokens in G, which is a contradiction. To see that inequality $(1 + (S_t \cdot D_t/(4 \cdot C)))^{d-1} \cdot c_T > 2 \cdot C$ holds, it suffices, by Lemma 3, to show that $(1 + (S_t \cdot D_t/(4 \cdot C)))^{d-1} > 2 \cdot n^3$. We prove the latter by obtaining the natural logarithms of both parts of the latter inequality, where $d = 32 \cdot C \cdot \ln n/(S_t \cdot D_t)+1$. After logarithmizing and since $4 \cdot \ln n > \ln(2 \cdot n^3)$, it suffices to show that

$$\frac{32 \cdot C \cdot \ln n}{S_t \cdot D_t} \cdot \ln\left(1 + \frac{S_t \cdot D_t}{4 \cdot C}\right) > 4 \cdot \ln n \ .$$

Let $\mu = D_t \cdot S_t/(4 \cdot C)$. We know that $D_t \cdot S_t < C$, and thus $\mu < 1$. We have that $1+\mu \leq e^{\mu} \leq 1 + \mu + \mu^2 < (1 + \mu)^2$. Therefore $2 \cdot \ln(1 + \mu) > \mu$ or, equivalently, $\ln(1 + \mu) > \mu/2$. We combine the above inequalities to show our goal. □

Lemma 4 combined with Eq. 10 leads to the following theorems:

Theorem 2. $\sum^i \text{dist}(P(s_i,t_i)) \cdot d(s_i,t_i) \leq \sum^i \text{length}(P(s_i,t_i)) \cdot d(s_i,t_i) = O(\log n \cdot \log l(V)) \cdot (l(V)/a)$.

Theorem 3. $a \leq a_A \leq O(\log n \cdot \log l(V)) \cdot a$.

Theorem 4. *We have that* $S/(64 \ln n \log D) \leq z^* \leq S$.

By Theorem 4 and since $l(V) \cdot S/2 \leq a \leq l(V) \cdot S$, we have that $l(V) \cdot z^*/2 \leq a \leq O(z^* \cdot l(V) \cdot \log l(V) \cdot \log n)$. Hence we approximate a by finding the optimal value of z^*. In order to find a cut that is an approximate flux, however, we follow the idea of Leighton-Rao Ref. 20 and Klein *et. al.* Ref. 10. We will first obtain an approximation S_{appr} of the minimum cut ratio S, i.e., we want to find a set $V_1 \subsetneq V$ for which $\sum_i \text{dist}_l(s_i,t_i) \cdot d(s_i,t_i) = O(\ln n \cdot \log D \cdot (d(\Gamma(V_1))/c(\Gamma(V_1)))$. Let S_{appr} be the cut ratio of the cut $\Gamma(V_1)$. The above equation when combined with Eq. 10 guarantees that $S_{appr}/O(\ln n \cdot \log D) \leq S$.

The idea for deriving the above equation is to substitute each S_t with a value that guarantees the above equation. Let S^t_{obs} be the observed value of S_t while the algorithm runs, i.e., the value of S^t_{obs} is the minimum value of $d(\Gamma(T))/c(\Gamma(T))$ over all sets appearing at stage t of the routing algorithm. The proof of Theorem 4 still holds if we replace the S_t of stage t by S^t_{obs}. Thus $S_{obs} = \min_t\{S^t_{obs}\}$ can serve as S_{appr}. (If S_{obs} occurs at some stage other than the initial, then the actual cut ratio for the cut is no more than S_{obs}. In this case, we set S_{appr} to the actual cut-ratio.)

However, we do not know the values of each S^t_{obs} before the algorithm completes the execution of stage t. But observe that if the value which substitutes S_t at stage t of the algorithm is no more than $(3/2) \cdot S^t_{obs}$, then we still have that the remaining demand at the next stage will be reduced by a factor of 3/4. Therefore we start the algorithm with some estimate of S_t derived from an arbitrary cut in G and we compare the resulting value of S^t_{obs} with the estimate when the stage terminates. If the estimate is less than or equal to $(3/2)\cdot S^t_{obs}$, then we are done and the cut for S^t_{obs} is put on a list of possible candidates for S_{appr}, otherwise we reduce the estimate by a factor of 2 and we repeat. We conclude that $S_{appr} \leq O(\log n \cdot \log D) \cdot S$. Now consider the edge expansion a_{appr} that corresponds to the cut which determines S_{appr}. From the definition of our concurrent flow problem we have that $a_{appr} \leq S_{appr} \cdot l(V)$. We combine all these as follows: $a_{appr} \leq S_{appr} \cdot l(V) \leq O(\log n \cdot \log l(V)) \cdot S \cdot l(V) \leq O(\log n \cdot \log D) \cdot a$. We conclude:

Theorem 5. *We can find a cut* $\Gamma(V_1)$ *with edge expansion* $O(\log n \cdot \log l(V)) \cdot a$.

4. Other Flux Approximation Algorithms

In this section, we briefly present graph and hypergraph approximation algorithms which, when combined with the BB algorithm described in Sec. 3, will lead to balanced graph bipartitions. The first algorithm described (Sec. 4.1) is due to Leighton and Rao[21] and guarantees an approximation when the graph has uniform weights on the nodes. The multicommodity and the concurrent flow problems used by Leighton and Rao are defined in the same way as the $\mathcal{GM}$ and $\mathcal{GC}$ flow problems of the previous section. The demands of the commodities are now equal. For example, each demand can be set to 1. The flow problems are now called the *Uniform Multicommodity* ($\mathcal{UM}$) and the *Uniform Concurrent* ($\mathcal{UC}$) flow problems respectively. Finally, in Sec. 4.2 we briefly discuss how the presented algorithms can be extended to approximate the flux in hypergraphs.

4.1. Algorithms for Graphs

Let $P(s_i, t_i)$ denote the path assigned to commodity i with source s_i and destination t_i, length$(P(s_i, t_i))$, be the length of this path subject to the length function ℓ. Leighton and Rao's algorithm, we call it LR, follows the same method as the flux approximation algorithm in Sec. 3. The goal is to find a cut $\Gamma(A)$ for which

$$\sum_i \text{length}(P(s_i,t_i)) \cdot d(s_i,t_i) = O\left(\left(\frac{l(V)}{a_A}\right) \cdot \log n\right) , \tag{14}$$

where a_A is the edge expansion of the cut $\Gamma(A)$. (See also Theorem 2 to observe the similarity of this approach to the one described in Sec. 3.) Also the term $\sum_i \text{length}(P(s_i,t_i)) \cdot d(s_i,t_i)$ in Eq. 14 can now be replaced by $\sum_i \text{length}(P(s_i,t_i))$ since every demand is 1. Moreover, algorithm LR guarantees a tighter bound than the bound of the algorithm in Sec. 3. That is, the routing of the commodities satisfies that:

$$\sum_i \text{length}(P(s_i,t_i)) = O\left(\left(\frac{l(V)}{a_A}\right) \cdot \log n\right) . \tag{15}$$

In algorithm LR we have the same discretization step as in the algorithm of Sec. 3. We also try to locate trees with small depth. However, the left hand side of Eq. 14 is the sum of the paths from the root of the tree to the remaining nodes of the tree. This observation allows us to modify the algorithm in Sec. 3 to obtain a tighter flux approximation. Leighton and Rao have proved in Ref. 21 that it suffices to locate only one tree that has at least half of the nodes in the graph and small depth. Then we again route the commodities with both endpoints on the tree. This is the main reason for saving the $\log l(V)$ factor on the approximation. In algorithm LR, described below, we use $\ell(e)$ and C to denote the length of edge $e \in E$ and the total capacity on the edges, respectively.

Algorithm LR

Step 1: Discretization of the edges

It is the same as the one described in algorithm in Sec. 3. However, in the original paper,[21] Leighton and Rao suggested that the lengths $\ell(e)$ be computed using Vaidya's linear programming algorithm[29] to obtain the $O(1-\epsilon)$ optimal approximation for the throughput z^* in the concurrent flow problem on B, where $0 < \epsilon < 1$. This part can be substituted with more efficient techniques. (See Sec. 6.)

Step 2: Expansion on a breadth first search tree using the token information

(a) Select an arbitrary node v in V to initiate a set of nodes in V. Call this node the *root* of the set.

(b) Extend the set by a level of tokens, expanding in a breadth first search manner. (The goal again is to create a breadth first search tree.)

(c) Let i be the depth of the breadth first search tree measured in number of tokens. Let C_i, a_{obs} be the total token capacity in the set at step i and the minimum observed edge expansion on G up to this step. If $C_i \geq (1+a_{obs} \cdot n/(9 \cdot C)) \cdot C_{i-1}$, go to Step 2(b).

(d) If the set contains half the nodes in V, then go to Step 3; Otherwise, remove from G all the nodes in the set together with the incident edges, update G, and go to Step 2(a).

Step 3: Completion of the tree construction and routing the commodities into paths
Proceed in a breadth first search manner, extending the set by adding a level of tokens at a time, at each step computing the minimum observed expansion a_{obs}, until all the nodes in V are reached. Keep expanding in a breadth first search manner on the tree which the previous step outputs. The approximate cut that the algorithm outputs is the minimum observed expansion a_{obs} at the end of the algorithm.

End of algorithm LR

Theorem 6.[21] *Algorithm LR finds an* $O(\log n)$ *approximation to the flux for graphs with uniform weights on the nodes.*

The approximation bound of algorithm LR holds only if the graph has uniform weights on the nodes. However, Leighton *et al.*[19] have recently improved the bound of the algorithm in Sec. 3 for general graphs, by appropriately modifying the LR algorithm. More precisely, Leighton *et al.*[19] showed that if we modify algorithm LR so that at Step 2(b) we select the node $l(v)$ with maximum node weight, then LR can locate a cut $\Gamma(A)$ for which the edge expansion a_A is at most $O(\log n)a_A$. We call the latter algorithm the *GLR* (*Generalized LR*) algorithm. We have the following theorem.

Theorem 7.[19] *Algorithm GLR obtains an* $O(\log n)$ *approximation of the flux a for any graph* G.

The proof is based on the fact that the demands of the commodities are not arbitrary but they are related to the weights on the nodes. This allows us to find only one breadth first search tree that satisfies:

$$\sum_i \text{length}(P(s_i,t_i)) \cdot d(s_i,t_i) = 2 \cdot \sum_{\forall v_1,v_2 \in V} \text{length}(P(v_1,v_2)) \cdot l(v_1) \cdot l(v_2)$$

$$= O\left(\left(\frac{l(V)}{a_A}\right) \cdot \log n\right) . \qquad (16)$$

From Eq. 16 we can easily observe that the technique is similar to the method described in Sec. 3. (Compare to Theorem 2.) This concludes the survey on existing flux approximation algorithms on graphs. It remains to analyze their complexities. This is the subject of Sec. 6. However, we briefly mention here that the LR algorithm is faster than the two existing approximation algorithms for general graphs.

4.2. Algorithms for Hypergraphs

Let us now consider the case when the input is a hypergraph $H = (V, E_h)$. The method is a little more complicated but the basic idea remains the same. Let $V_1 \subset$

V define a bipartition of H. The flux a of H is defined similarly to the way that it is defined on a graph G. Namely,

$$a = \min_{V_1 \subset V} \left\{ \frac{\sum^{e \in \Gamma(V_1)} w(e)}{\min\{l(V_1), l(V - V_1)\}} \right\} . \tag{17}$$

In order to approximate the flux of H, we first need to model H as a bipartite graph. The *bipartite graph* $B = (V, A, E)$ of H is constructed as follows: Each node v in A corresponds to hyperedge v in E_h and an edge (w, v) is in E if and only if hyperedge v in E_h is incident to node w in V. Let m_b denote the number of edges in B. Clearly, m_b is $O(n \cdot m)$ and the number of nodes in B is $O(n+m)$. The weight $l(v)$ of each node $v \in V$ is equal to the weight of the respective node in H. The weight $l(v)$ of each node v in A is equal to the weight $w(v)$ of the hyperedge v in E_h. Any edge $e \in E$ incident to node $v \in A$ has weight $w(e) = l(v)$. (See Fig. 1.) The bipartite graph B is called the *bipartite graph representation* of H. Any bipartitioning problem on H is now interpreted on B as follows: Partition the nodes of V into sets V_1 and $V - V_1$ so that the total weight on the nodes in A, which are adjacent to at least a node in V_1 and at least anode in $V - V_1$ is minimized.

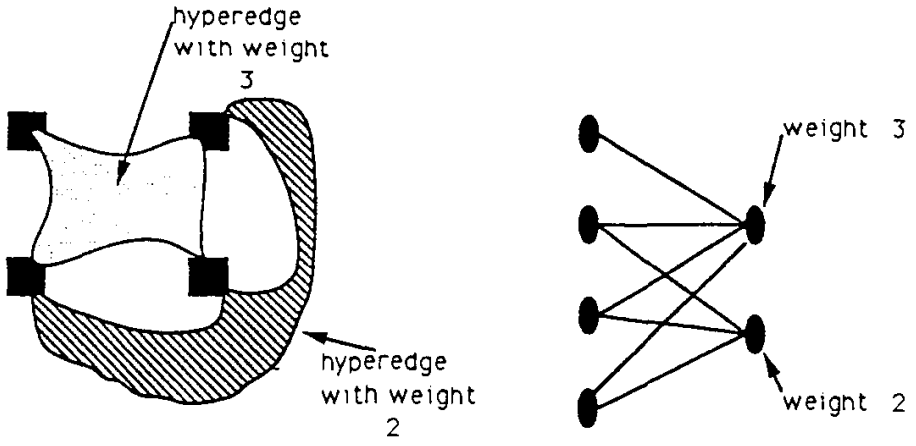


Fig. 1. H and its bipartite graph representation B.

Following the basic idea for the approximation of the flux in graphs, we will define a quantity a_b that represents the flux of H on B and a multicommodity flow problem on B such that a_b and the minimum cut ratio S on B (see Eq. 4) are within a constant factor of each other. Let T be the set that consists of all nodes in A which are incident to edges in the cut $\Gamma(V_1)$ defined by a set $V_1 \subset V$. We define on B a quantity a_b, the *bipartite flux*, which naturally represents the flux a of H on the bipartite graph. More precisely, a_b is:

$$a_b = \min_{V_1 \in V} \frac{\sum^{v \in T} c(v)}{\min\{l(V_1), l(V - V_1)\}} . \tag{18}$$

The quantity a_b is interpreted as follows: B has flux a_b if every set V_1 with at most half of the nodes in V is connected to the set V-V_1 via nodes in A whose total capacity is at least $a_b \cdot l(V_1)$. Clearly, we can approximate the flux a on H, by approximating a_b on B.

Next, we define the concurrent flow problem on the bipartite graph $B = (V, A, E)$. We call this flow problem the *hypergraph multicommodity* ($\mathcal{HM}$) flow problem. Its corresponding optimization version is called the *hypergraph concurrent* ($\mathcal{HC}$) flow problem. In the $\mathcal{HM}$ flow problem there are capacity constraints $c(v) = l(v)$, $\forall v \in A$, and $c(e) = w(e)$, $\forall e \in E$. Each commodity is defined by a directed pair (v, w) of nodes, $\forall v, w \in V$. The demand $d(v, w)$ for the commodity from node v to node w in V equals $(l(v) \cdot l(w))/2$.

Consider now the minimum cut ratio S for the $\mathcal{HM}$ flow problem on B. (A cut can now be formed by selecting nodes as well.) The weights on the edges $e \in E$ enable us to the following definition of the minimum cut ratio S on B.

$$S = \min_{V_1 \subset V} \left\{ \frac{\sum^{v \in T} c(v)}{l(V_1) \cdot l(V - V_1)} \right\} . \tag{19}$$

Using similar arguments to the ones in Sec. 2.2, we have that the minimum cut ratio S and the bipartite flux a_b satisfy Eq. 6.[23,18] The remaining of the technique resembles the approach described in the previous sections for graphs. We apply duality arguments, as we did in Sec. 3 for graphs, and we obtain a length for each node $v \in A$ and each edge $e \in E$. (This involves a transformation of the bipartite graph B to a digraph so that we can handle the capacities on the nodes $v \in A$. The details are omitted here.) In Refs. 23, 27, 18, we give an algorithm that finds a cut $\Gamma(V_1)$, $V_1 \subset V$, with hyperedge expansion a_{V_1} such that

$$\sum^{i} \text{dist}_{\ell}(s_i, t_i) \cdot d(s_i, t_i) = O\left(\frac{\log(\max\{m, n\})}{a_{V_1}} \right) , \tag{20}$$

where now the shortest path $\text{dist}_{\ell}(s_i, t_i)$ for commodity i is measured in terms of the lengths $\ell(v)$ and $\ell(e)$ assigned on the nodes $v \in A$ and the edges $e \in E$, respectively. The above equation guarantees that the set of nodes V_1 has a hyperedge expansion which is within an $O(log(\max\{n, m\}))$ factor of the flux a, where m, n are the number of hyperedges and nodes, respectively. In Refs. 27, 23, this was shown for hypergraphs with uniform weights on the nodes. A generalization to general hypergraphs is given in Ref. 18. The description of these algorithms is omitted. Their time complexities are also determined by the $O(1-\epsilon)$ approximations for the $\mathcal{HC}$ flow problem and the techniques described in Sec. 6 can be modified to apply to the $\mathcal{HC}$ problem.

5. Approximation Algorithms for Multiway Partitioning

Consider a partition of the input graph $G = (V, E)$ (or hypergraph $H = (V, E)$) into sets $V_1, \ldots, V_p$. Let p, n, m be the number of sets in the partition and let k_i be

the size of the set V_i, the nodes of G (or H), the edges (or hyperedges) of G (or H), respectively. In multiway graph or hypergraph partitioning we have that $3 \leq k_i \leq l(V)/3$. As mentioned in Sec. 1, depending on the formulation of the optimization function, we distinguish between two versions of the multiway partitioning problem on hypergraphs which differ on the objective function to be minimized. (See also Sec. 1.)

In this section, we briefly present approximation algorithms for graph and (both versions of) hypergraph multiway partitioning. (The algorithms and their proof of optimality are given in detail in Refs. 17, 18.) The algorithms are based on the balanced bipartitioning BB algorithm by Rao (see Sec. 2.1) and will be presented for the case when the input is a hypergraph $H = (V, E)$ with arbitrary integer weights on the nodes and the edges. We give two algorithms: The first obtains a better approximation bound than the second but it has higher time complexity. Furthermore, the first algorithm can also be applied for both versions of multiway partitioning while the second guarantees approximation bounds only for Version 2.

We call the first algorithm *MHP* (*Multiway Hypergraph Partitioning*) and the second *AMHP* (*Alternative Multiway Hypergraph Partitioning*). The basic component of algorithm *MHP* is a subroutine that obtains approximate balanced bipartitions in two steps. In the first step we call the flux approximation algorithm of Sec. 4.2 to approximate the flux. In the second step we use the balanced bipartitioning algorithm of Sec. 2 to result in an approximate balanced bipartition of the input hypergraph. Now the flux and the balanced bipartition obtained are approximated within a $O(\log(\min\{n, m\}))$ factor of the optimal.[23,27] The *AMHP* algorithm differs from the *MHP* algorithm in that it first transforms the hypergraph into a graph and then uses the balanced bipartitioning algorithm in Sec. 2.1 on the generated graph. Moreover, the balanced bipartitioning algorithm uses the $O(\log n)$ times optimal *GLR* approximation algorithm given in Ref. 19. (See also Sec. 4.1). The *AMHP* algorithm is faster than the *MHP* algorithm because the flux approximation subroutine requires a $O(1-\epsilon)$ approximation for the $\mathcal{GC}$ flow problem, $0 < \epsilon < 1$. (The *MHP* algorithm requires an $O(1-\epsilon)$ approximation for the $\mathcal{HC}$ flow problem which is more time consuming. See Sec. 6.) However, the bound obtained by algorithm *AMHP* is inferior to the one obtained by algorithm *MHP*. In the two subsections below we describe the two algorithms in more detail.

5.1. The MHP Algorithm

The idea is to call algorithm BB (Balanced Bipartitioning) recursively until we result to the required set sizes. At each step of the iteration we have a collection of sets, and we apply the BB algorithm to all of these sets with size more than k. The cost of the iteration equals to the sum of the weights on the edges on cuts encountered during the iteration. Based on the values of the weights on the nodes of the input hypergraph $H = (V, E)$, we first select proper values for α and β to guarantee that we can recursively obtain $\alpha\beta - BB$s on the induced subgraphs. (This is not needed if the hypergraph has uniform weights on the nodes.)

Assume that the values of α and β are such that at each iteration we can obtain a 4-balanced bipartition which is within a logarithmic factor to the optimal 3-balanced bipartitioning. We prove the approximation bound by showing that the cost of each iteration is not more than the optimal partitioning into the required set sizes. The algorithm distinguishes between cases depending on the size of k. If k is small enough, the approximation bound is better. Algorithm *MHP* is given below:

Algorithm Multiway Hypergraph Partitioning (MHP)

Distinguish between two cases: (a) $k > \log n$ and (b) $k \leq \log n$.

Step 1(a) $k > \log n$: Apply algorithm *BB* recursively to obtain a balanced bipartitioning on any induced hypergraph with size more than k. If an induced hypergraph has size less than or equal to k, stop the recursion and output the partition.

Step 1(b) $k \leq \log n$: Apply algorithm *BB* recursively on each induced hypergraph with size more than $4 \log n$. Now each resulting hypergraph $H_i = (V_i, E_i)$ has size in the range $[\log n, 4 \log n]$. This allows us to partition every H_i into sets of size at most k optimally, by using, for example, a brute force exponential algorithm. The latter takes $O(n)$ time.

End of Multiway Hypergraph Partitioning (MHP)

Let APP_k, OPT_k and $\mathrm{OPT}_{k/3}$ denote a partition of algorithm *MHP* into sets of size at most k, an optimal partition into sets of size at most k and an optimal partition into sets of size at most $k/3$, respectively. We have:

Theorem 8.[17,27] *For all* k, $|\mathrm{APP}_k| \leq O(\log m \cdot \log(l(V)/k))|\mathrm{OPT}_{k/3}|$. *If* $k \leq \log n$, *then* $|\mathrm{APP}_k| \leq O(\log m \cdot \log l(V))|\mathrm{OPT}_k|$.

We sketch the proof of Theorem 8 by using a lemma proved in Refs. 27, 18. Suppose that the set of nodes V of the input hypergraph has been partitioned into p sets $V_1, \ldots, V_p$, each of size no more than k. Let set V_i have n_i nodes, and size $l(V_i)$. Let OPT_3^i, OPT_k be the optimal 3-balanced bipartition for the set V_i, and an optimal partition of H into sets of size at most k. We have:

Lemma 5.[17,18] *If* $k \leq \min\{l(V_1)/3, \ldots, l(V_p)/3\}$, *then* $|\mathrm{OPT}_k| \geq \sum_{i=1}^{p} |\mathrm{OPT}_3{}^i|$.

The lemma states that the cost of an iteration of the *MHP* algorithm is not more than the optimal partition into sets of size at most k. Observe that Lemma 5 holds for all sets, at each level of the partition tree which is formed while applying the *BB* algorithm recursively. In case (*a*) of algorithm *MHP* there are $O(\log(l(V)/k))$ levels in the partition tree. By Lemma 5 and the approximation bound that algorithm *BB* obtains, we have that $|\mathrm{APP}_k| \leq O(\log m \cdot \log(l(V)/k)) \cdot |\mathrm{OPT}_{(k/3)}|$. In case (*b*) of algorithm *MHP* we have a more elegant bound. Now there are $O(\log l(V))$ levels in

the partition tree. In the last step of the algorithm, we remove hyperedges optimally. This is sufficient to show that in this case $\mathrm{APP}_k \leq O(\log l(V) \cdot \log m) \cdot |\mathrm{OPT}_k|$.

A detailed proof of Theorem 8 appears in Ref. 27. Also recently[18] we modified the algorithm to obtain a tigher bound. Namely, in the first case the bound is improved to $O(\log m \cdot \log(n/k))$; if $k \leq \log n$, it becomes $O(\log m \cdot \log n)$.

5.2. Algorithm AMHP

Algorithm *MHP* of the previous section is based on the transformation of H into its bipartite graph B. The bipartite graph B has $O(m \cdot n)$ edges and $O(m + n)$ nodes. In Sec. 6 we show that the solutions to the $\mathcal{HC}$ flow problem on B are slower than the solutions of the $\mathcal{GC}$ flow problem on a graph G with n nodes and m edges. Algorithm *AMHP* uses a transformation to a graph with n nodes and m edges to result in faster but not as (provably) good solutions. Namely, the approximation bounds that *AMHP* obtains will be nontrivial if either k (the maximum set size), or r (the maximum hyperedge size) is polylogarithmic to the input size.

Algorithm Alternative Multiway Hypergraph Partitioning (AMHP)

Step 1: Removing hyperedges:
Given $H = (V, E_h)$, remove all the hyperedges that connect more than k nodes. Let D_1 be the set of the removed hyperedges. Let $H_1 = (V, E'_h)$ be the resulting hypergraph. Observe that the hyperedges in D_1 will always be encountered in any optimal solution for the second version of the multiway hypergraph partitioning problem.

Step 2: Graph representation:
Represent H_1 by a graph G_h, in which every hyperedge that connects $r > 2$ nodes is substituted by a chain of length $r - 1$ that connects these r nodes. Merge all possible multiple edges into one weighted edge.

Step 3: Deriving partitions:
Apply the algorithm for multiway graph partitioning algorithm on G_h. (The corresponding of algorithm *MHP* for graphs.[17,27]) This results into sets of size at most k. Compute the number of hyperedges that connect nodes in different sets. Let D_2 be the set of these hyperedges. The hyperedges in D_1 together with the hyperedges in D_2 form the solution to our partitioning problem.

End of algorithm Alternative Multiway Hypergraph Partitioning

Let APP_k be the solution obtained by this algorithm. Let OPT_k and $\mathrm{OPT}_{k/3}$ be the optimal solutions for partitioning H into sets of size at most k and $k/3$, respectively. Let r be the number of nodes in the largest hyperedge of H, that is the hyperedge with the largest size. The following theorem states the approximation bound of algorithm *MHP*.

Theorem 9.[17,18] $|\mathrm{APP}_k| = O(\min\{k,r\}\cdot\log n\cdot\log l(V)\cdot\log l(V)/k))\cdot|\mathrm{OPT}_{k/3}|$. *If* $k \le \log n$, *then* $|\mathrm{APP}_k| = O(\min\{k,r\}\cdot\log n\cdot(\log^2 l(V)))\cdot|\mathrm{OPT}_k|$.

The proof of the theorem is described in detail in Ref. 27. Recently, in Ref. 18 we modified the algorithm to obtain a bound within a $O(\min\{k,r\}\cdot\log^2 n\cdot\log(n/k))$ factor. In the special case when $k \le \log n$, the bound becomes $O(\min\{k,r\}\cdot\log^3 n)$. Both *MHP* and *AMHP* algorithms can be modified to obtain approximations (within the same factors of the optimal) for the equal size multiway partitioning problem.[27]

6. Fast Implementations of the Partitioning Algorithms

As mentioned in Secs. 2.2 and 3, the flux approximation algorithms require an approximation within a $O(1-\epsilon)$ factor of z^* in the $\mathcal{GC}$ and $\mathcal{UC}$ flow problems on graphs and the similarly defined $\mathcal{HC}$ and *Uniform Hypergraph Concurrent* $\mathcal{UHC}$ on hypergraphs. (The $\mathcal{UHC}$ flow problem is used for the special case of hypergraphs with uniform weights on the nodes and it is defined similarly to the $\mathcal{HC}$ flow problem but now each demand is 1.[27])

Let K, n, m denote the number of commodities, the number of nodes and the number of edges in the graph G on which we define the concurrent flow problem, respectively. As we mentioned in Sec. 1, an $O(1-\epsilon)$ approximation to any concurrent flow problem can be obtained using linear programming. Let $l_{\max}$, $w_{\max}$ be the largest weight on any node and edge of G, respectively. The linear programming formulation will result to an $\tilde{O}(n^7\cdot m^{.5})$ time algorithm for approximating the flux on graphs. This is obtained because in the $\mathcal{GC}$ concurrent flow problem the number of commodities $K = O(n^2)$. We can reduce, however, the number of commodities to $O(n)$ if we are willing to approximate the flux with very high probability. In this case the time complexity is $\tilde{O}(n^{4.5}\cdot m^{.5})$. In the flux approximations, however, ϵ can be close to 1. (For example we can set $\epsilon = 1/2$.) Therefore the time complexities do not depend on ϵ^{-1}. Moreover, Rao's bipartitioning algorithm (see Sec. 2.1) requires an additional $O(n)$ factor and the multiway partitioning algorithms an additional $O(n\cdot\log n)$ factor.

It is therefore important to obtain faster flux approximations that use arguments other than linear programming for approximating the flow problems. The first combinatorial $O(1-\epsilon)$ approximation algorithms for concurrent flow problems are presented by Sharhokhi and Matula[26] and are based on flow rerouting techniques. Klein *et al.*[14] refined the technique in Ref. 26. However, both algorithms handle only the special case of concurrent flows with uniform demands.

In this section, we describe fast combinatorial flux approximations for graphs and hypergraphs described in Secs. 3 and 4. We call the algorithm that approximates the $\mathcal{GC}$ and $\mathcal{HC}$ flow problems the *General Demand Flow* (*GDF*) algorithm. As we mentioned in Sec. 4, the difference between the $\mathcal{GC}$ flow problem on G and the $\mathcal{HC}$ flow problem on B is that in $\mathcal{GC}$ we have capacity constraints on the edges while in $\mathcal{HC}$ the capacity constraints are on the nodes of the A set of the bipartite graph B. In Ref. 27 we describe a transformation of graph B to a directed graph G' with

capacities only on the edges. Since algorithm *GDF* can also approximate concurrent flow problems on digraphs, we derive an approximation to the $\mathcal{HC}$ problem using the *GDF* algorithm as well. (However, this is more time consuming due to the size of graph G' which has $O(m \cdot n)$ edges and $O(n+m)$ nodes.)

For the special case when the nodes have uniform weights, the $\mathcal{UC}$ and $\mathcal{UHC}$ flow problems have uniform demands. We can approximate these problems faster than the $\mathcal{GC}$ and $\mathcal{HC}$ flow problems using an alternative algorithm which we call the *Unit Demand Flow* (*UDF*) algorithm.

In Sec. 6.1 we describe the *GDF* algorithm and in Sec. 6.2 we give the *UDF* algorithm. The algorithms are presented for the $\mathcal{GC}$ and the $\mathcal{UC}$ flow problems on an input graph $G = (V, E)$ with capacity $c(e)$ on every edge e and demands $d(i) = d(s_i, t_i)$, for every pair of s_i and $t_i \in V$. Then we give the time complexities for the flux approximation algorithms presented in Secs. 3 and 4. Finally, in Sec. 6.3 we give the time complexities for the balanced bipartitioning and multiway partitioning algorithms presented in Secs. 2 and 5, respectively.

Throughout this section we will use z^*, K, $c_{\max}$, $d_{\max}$ to denote the maximum demand percentage satisfied, the number of commodities, the maximum capacity constraint on the edges, and the maximum demand, respectively. The basic idea, for both algorithms described in this section, is an equivalent problem formulation. Consider a routing of the commodities which satisfies the demand for each commodity i. We define λ to be the minimum among all edges ratio: flow through the edge divided by the capacity of the edge. We call λ the *capacity utilization*. The goal now in both $\mathcal{GC}$ and $\mathcal{UC}$ is to reroute the commodities so that we minimize λ. Minimizing λ, in this new problem formulation, is equivalent to maximizing z. We use λ^* to denote the minimum λ among all possible routings of the commodities. We have that $\lambda^* = 1/z^*$. We call λ^* the *minimum capacity utilization*.

In the $\mathcal{GC}$ and $\mathcal{UC}$ flow problems we are seeking for an $O(1-\epsilon)$ approximation of the throughput z^*. In the equivalent problem formulation given above, we want to find a multicommodity flow f satisfying the demands $d(i) = d(s_i, t_i)$ so that λ is at most a factor $(1+\epsilon)$ more than λ^*. In this case, we say that the multicommodity flow f is ϵ *− optimal*.

6.1. Non Uniform Demands: Algorithm GDF

Algorithm *GDF* finds an $O(1-\epsilon)$ approximation for the throughput z^* of the $\mathcal{GC}$ flow problem on graph $G = (V, E)$. More precisely, it finds a multicommodity flow f that satisfies demands $d(s_i, t_i)$ so that λ is at most a factor $(1+9\epsilon)$ more than λ^*. Equivalently, the flow f is 9ϵ-optimal.* Algorithm *GDF* can also handle the more general flow problem on G where we are allowed to have K arbitrary commodities and each commodity is defined by a set of sources and a set of destinations. Observe in $\mathcal{GC}$ we have that $K = n \cdot (n-1)$.

*In Ref. 19, the authors show that the algorithm obtains $O(1-\epsilon)$ approximations for any $0 < \epsilon < 1$. In this more general case, the time complexity of the solution also depends on ϵ^{-1}.

The outline of the *GDF* algorithm is as follows: Initially, the algorithm finds a good initial solution for the *GC* flow problem. Then it calls a procedure DECONGEST which takes a flow which satisfies capacity utilization λ and produces a new flow which is either 9ϵ-optimal or has capacity utilization at most $\lambda/2$. Procedure DECONGEST involves minimum cost flow computations. We use the cost-scaling algorithm of Goldberg and Tarjan[7] to compute the minimum cost flows. A formal description of the *GDF* algorithm follows:

The General Demand Flow (GDF) Algorithm

Step 1: Find an Initial Solution: Merge all commodities that share a source into a single commodity. Let K^* denote the number of commodities in this new problem. (Thus for the *GC* problem $K^* = n$.) Find an initial flow, by routing each commodity i, $1 \leq i \leq K^*$, separately and using standard max-flow techniques. This results to a flow f satisfying demands such that $\lambda \leq K \cdot \lambda^*$. (The proof of the latter is given in Refs. 27, 19.)

Step 2: Reroute Flow: In order to produce a new flow which is closer to the optimal, call procedure DECONGEST (described below). DECONGEST takes a flow f with utilization λ_0 and produces a new flow f' which is either 9ϵ-optimal or has $\lambda \leq \lambda_0/2$.

Step 3: Compute the 9ϵ-optimal flow: Repeat Step 2 until an $O(\epsilon)$ flow is encountered.

End of algorithm General Demand Flow (GDF)

At this point we would like to mention that although algorithm *GDF* produces an $9 \cdot \epsilon$-optimal flow and a length function $\ell'(e)$ for every edge $e \in E$, the computed length function ℓ' cannot be directly used in neither the algorithm of Sec. 3 nor the *GLR* algorithm. It needs to be modified to a new applicable length function $\ell(e)$.

Recall Eq. 12 which is the goal of the two approximation algorithms. As we have already mentioned, in order to be able to replace the optimal linear programming concurrent flow computation with a more efficient approximate computation, we need to compute a length function $\ell(e)$ such that for a constant $\epsilon > 0$, this function satisfies:

$$\frac{\sum_{s,t \in V} d(s,t) \cdot \text{dist}_\ell(s,t)}{\sum_{e \in E} \ell(e) \cdot c(e)} \geq \frac{\lambda^*}{1+\epsilon} \,. \tag{21}$$

Unfortunately, the function ℓ computed by algorithm *GDF* does not guarantee Eq. 21 above. In algorithm *GDF* the nominator of the ratio on the right hand side equals to a quantity which represents the cost of the routing of the commodities given the length function ℓ'. We use $Q_i^*(\lambda) = \sum_e |f_i^*(e) \cdot |\ell'(e)$ to represent this cost, where f_i^* is a flow that satisfies the demands for commodity i subject to the capacity constraints $\lambda \cdot c(e)$ on each edge e. In Ref. 19 we show how to modify function ℓ' to a function ℓ that satisfies Eq. 21.

Next, we proceed with the description of procedure DECONGEST. In DECONGEST we reroute commodities. The procedure iteratively computes f_i^* and minimizes $Q_i^*(\lambda)$. Namely, DECONGEST takes a flow f with utilization λ_0 and produces a new flow f' which is either $9 \cdot \epsilon$-optimal or has $\lambda \leq \lambda_0/2$. The basic idea is that the procedure reroutes an appropriately chosen fraction of flow of a "bad" commodity onto the edges of a minimum-cost flow associated with this commodity (as described below), in order to decrease congestion. The distinction of a commodity as "bad" or not is based on distance criteria and therefore on the definition of a length function. The length function used in DECONGEST is $\ell'(e) = \epsilon^{\alpha \cdot \lambda(e)}/c(e)$, where the value of α depends on the current congestion λ, on the constant ϵ and on the number of edges m. Let Q_i be the cost of the current flow for commodity i, using ℓ' as the cost function. A "bad" commodity is a commodity that satisfies the following inequality: Q_i-$Q_i^*(\lambda) > \epsilon \cdot Q_i + \epsilon \cdot \lambda / K^* \cdot \sum_{e \in E} c(e) \cdot \ell'(e)$.

In each iteration, we first choose a "bad" commodity i, and formulate an auxiliary transshipment problem. The demand of each node v in the auxiliary problem is equal to $\hat{d}_i(v)$, i.e., the set of demands for commodity i. The desired flow $f_i^*(e)$ is constrained to be between $-\lambda \cdot c(e)$ and $\lambda \cdot c(e)$, where λ is the current capacity utilization. By rerouting flow we try to minimize $Q_i^*(\lambda) = \sum_{(v,w)} |f_i^*(v,w)| \cdot \ell'(v,w)$. Let $\alpha = 2 \cdot (1+\epsilon) \cdot \lambda^{-1} \cdot \epsilon^{-1} \cdot \ln(m \cdot \epsilon^{-1})$. Given an optimal solution to this auxiliary problem, we reroute a $\sigma = \epsilon/(4 \cdot \alpha \cdot \lambda)$ fraction of the flow f_i onto the flow-paths of f_i^* by setting $f_i(e) \leftarrow (1-\sigma) \cdot f_i(e) + \sigma \cdot f_i^*(e)$, we recompute the length function, and we repeat. Upon termination, DECONGEST returns an improved flow f which is either $9 \cdot \epsilon$-optimal or has maximum congestion $\lambda \leq \lambda_0/2$. In Refs. 14, 19, the authors show that the above is guaranteed by the fraction of flow σ selected each time for rerouting in DECONGEST. Procedure DECONGEST is more formally described as follows[19]:

Procedure DECONGEST (f, ϵ)

$\alpha \leftarrow 2 \cdot (1+\epsilon) \cdot \lambda^{-1} \cdot \epsilon^{-1} \cdot \ln(m \cdot \epsilon^{-1})$; $\sigma \leftarrow \frac{\epsilon}{4 \cdot \alpha \cdot \lambda}$; $\lambda_0 \leftarrow \lambda$
While $\lambda \geq \lambda_0/2$ **and** f and ℓ' are not $9 \cdot \epsilon$-optimal
  For each edge (e), $\ell'(e) \leftarrow \epsilon^{\alpha \cdot \lambda(e)}/c(e)$.
  Find a "bad" commodity i.
  Formulate an auxiliary transshipment problem on G, where each node v has demand $\hat{d}_i(v)$, and the flow on edge vw is constrained to be between $\lambda \cdot c(e)$ and $\lambda \cdot c(e)$.
  Compute flow f_i^* that minimizes $Q_i^*(\lambda) = \sum_{vw} |f_i^*(e)| \cdot \ell'(e)$ in the auxiliary problem.
  For all $e \in E$, $f_i(e) \leftarrow (1-\sigma) \cdot f_i(e) + \sigma \cdot f_i^*(e)$.
Return f

End of procedure DECONGEST

Now we discuss the time complexity of DECONGEST. The computation intensive part of DECONGEST is finding a "bad" commodity and computing minimum-

cost flows. We can compute the lengths on the edges in $O(m \cdot \log n)$ time.[20] All the rest can be done in $O(m)$ time.[20] We find a "bad" commodity by comparing the costs $Q_i = \sum_{e \in E} |f_i(vw)| \cdot \ell'(e)$ and the costs of the minimum-cost flows. In the worst case, we need to check all K^* commodities. Hence, an iteration can be implemented in the time it takes to perform K^* minimum-cost flow computations.

We can perform this computation more efficiently by using a simple randomized strategy proposed initially in Ref. 13. If we compute the cost Q_i of each commodity and then randomly choose a commodity with probability proportional to its cost, then with probability of at least ϵ, we have chosen a "bad" commodity. By computing a single minimum-cost flow we can check whether the commodity is indeed "bad". We expect to perform this computation ϵ^{-1} times, and hence an iteration can be implemented in expected time equal to ϵ^{-1} times the time to perform a minimum-cost flow calculation. (Observe that the time to compute the cost of all current flows is dominated by the time to compute a minimum-cost flow). After every K^* iterations we can compute minimum-cost flows associated with all the flows and determine whether the current flow is $9 \cdot \epsilon$-optimal. Therefore, we can implement DECONGEST as a Las-Vegas algorithm. Note that this results in at most a factor of two increase in the number of minimum-cost flows computed during the execution of DECONGEST.

To conclude with the description of DECONGEST we briefly present the approximate min-cost flow routine which DECONGEST uses to compute minimum cost flows. The following algorithm, we call it the *GC* (*General Cost*) algorithm, computes only an approximately minimum-cost flow for a commodity i. In Ref. 19, it is shown that the approximation obtained is sufficient for the selection of a bad commodity.

Algorithm General Cost (GC)

Step 1. Round the demands for commodity i down to multiples of $\mu' = (\epsilon \cdot \lambda)/(4 \cdot n \cdot K^*)$.

Step 2. Delete all edges with costs less than Q_i/μ'. Then use the approximate Goldberg-Tarjan algorithm[7] on the problem with the rounded demands.

Step 3. Satisfy the remaining demands by assigning arbitrary residual paths from nodes with excess to nodes with deficit.

End of algorithm General Cost (GC)

This concludes the abstract description of the *GDF* algorithm whose time complexity is stated in the following theorem. (For the proof of this theorem as well as for the remaining theorems of this section, the reader is referred to Ref. 19.)

Theorem 10. *Algorithm GDF obtains an ϵ-optimal flow for the GC flow problem with $K^* = n$ commodities in $O(K^{*2} \cdot \log n \cdot \log K^* \cdot n \cdot m \cdot \log(c_{\max} \cdot n) \cdot \log(n^2/m))$ time. If we choose the "bad" commodities with the described randomized strategy the time complexity becomes $O(K^* \cdot \log n \cdot \log K^* \cdot n \cdot m \cdot \log(c_{\max} \cdot n) \log(n^2/m))$.*

From the description of the flux approximation algorithms in Secs. 3 and 4 it is not difficult to extract that the flow computation is the computationally intensive part. Let G be a graph of n nodes and m edges.

Theorem 11. *Algorithm GLR obtains an edge expansion which is within a* $O(\log n)$ *factor of the flux a in* $O(n^3 \cdot m \cdot \log^2 n \cdot \log(c_{\max} \cdot n) \cdot \log(n^2/m))$ *time. Alternatively, a randomized computation requires in* $O(n^2 \cdot m \cdot \log^2 n \cdot \log(c_{\max} \cdot n) \cdot \log(n^2/m))$ *time.*

The randomized version is due to the randomized strategy which we apply to select a "bad" commodity. The same complexity analysis holds for the algorithm described in Sec. 3. Although we have not described in detail the corresponding version of *GLR* (or *FA*) approximation algorithm for hypergraphs, we give its time complexity. (See also Ref. 27.) We call this algorithm the HYPERGRAPH FLUX algorithm.

Theorem 12. *Algorithm HYPERGRAPH FLUX obtains an edge expansion which is within a* $O(\log(\max\{n, m\}))$ *factor of the flux in* $O((n+m)^3 \cdot m_b \cdot \log^2 \cdot (n+m) \cdot \log(c_{\max} \cdot (n+m)) \cdot \log((n+m)^2/m_b))$ *time. Alternatively, a randomized computation can be obtained in* $O((n+m)^2 \cdot m_b \cdot \log^2 \cdot (n+m) \cdot \log(c_{\max} \cdot (n+m)) \cdot \log((n+m)^2/m_b))$ *time.*

Theorem 12 is derived from Theorem 11 if we substitute m by $m \cdot n$ and n by $n + m$. This is due to the transformation of the hypergraph to a graph on which we define the flow problems.

6.2. Uniform Demands: Algorithm Unit Demand Flow (UDF)

In this section we describe algorithm *UDF* for the *UC* flow problem. The algorithm is faster than the *GDF* algorithm of the previous section. The description of the *UDF* algorithm will be high level and we will not proceed to any proofs. We refer the interested reader to Refs. 19, 27. The time complexity of the algorithm depends on d, the maximum degree of any node in the input graph G. Algorithm *UDF* consists of three steps. In the first step we reduce the number of commodities using an approach different from the one described in algorithm *GDF*. In the second step we find an initial flow. In the third step the algorithm calls procedure DECONGEST as a subroutine which, however, uses a different procedure for computing approximately minimum cost flows. We give a more formal description of the *UDF* algorithm below:

Algorithm Uniform Demand Flow (UDF)

Step 1: Reduce the number of commodities: Randomly choose $K = O(n)$ source-sink pairs to create the set of commodities for the new concurrent flow problem on which we will work further. In the new flow problem, we call

it $\mathcal{MUD}$ (*Modified Uniform Demand*) flow problem, every node will be either a source or a sink in exactly three commodities.[19,18] It has been shown[20] that an ϵ-optimal flow for the $\mathcal{MUD}$ flow problem guarantees an $O(\epsilon)$-optimal flow for the $\mathcal{UD}$ flow problem. Let $G' = (V', E')$ be a graph created from the $\mathcal{MUD}$ flow problem as follows: Each node $v \in V'$ corresponds to a commodity source source or a commodity sink and each edge (v_1, v_2) has weight equal to the demand of the commodity from node v_1 to node v_2. Let c be the expansion parameter of G'.

Step 2: Find an initial flow: Find an initial flow that is optimal up to a factor of $O(m_b \cdot K)$ by routing each demand on the path with maximum bottleneck capacity from its source to its sink.

Step 3: Solve the *MUD* flow problem:

(a) Round the capacities up to integer multiples of $\epsilon \cdot c/(10 \cdot \lambda \cdot \epsilon \cdot \log^2 n)$.

(b) Call DECONGEST and solve the minimum-cost flows approximately as in algorithm *GDF*, but now substitute algorithm *GC* with Ford and Fulkerson's[5] algorithm. Use the flow obtained in the previous iteration as the initial flow for a new iteration.

(c) Repeat Steps 3(a) and 3(b)until an ϵ-optimal flow results.

End of algorithm Uniform Demand Flow (UDF)

Let d be the maximum node degree of the input graph G. Theorem 13 below gives the time complexity of algorithm *UDF*. A similar theorem can be derived for the case of hypergraphs by substituting m by $m \cdot n$ and n by $m + n$.

Theorem 13.[19] *For any constant ϵ, the randomized UDF algorithm obtains an $O(\epsilon)$-optimal flow for the UC flow problem in $O(n \cdot m \cdot d \cdot \log^4 \cdot n \cdot \log(n \cdot c_{\max}))$ expected time.*

Therefore the time complexity of the *LR* algorithm which computes the flux on uniform weighted graphs is now faster. This is given by the following theorem:

Theorem 14.[19] *The LR algorithm approximates the flux in $\tilde{O}(n \cdot m \cdot d)$ expected time.*

6.3. The Time Complexities of the Partitioning Algorithms

This section summarizes the time complexities of the partitioning algorithms. These complexities are determined by the subroutine that approximates the flux. The balanced graph and hypergraph bipartitioning algorithms have $O(\delta \cdot n)$ time complexity. Let δ be the time complexity described in Theorems 11, 12 and 14, respectively. The complexity of the multiway graph and hypergraph partitioning algorithms is $O(\delta \cdot n \cdot \min\{n, \log(l(V))\})$. If $k \leq \log n$, the time complexity becomes $O(\delta \cdot n \cdot \log(\min\{n, \log(l(V))\})$.

The algorithms for hypergraph partitioning are slower than the corresponding graph algorithms of the same input size. This is due to the bipartite graph representation of the input hypergraph H. In multiway hypergraph partitioning, however, we can avoid this time overhead by using the *AMHP* algorithm. Let $c_{\max}$ be the maximum capacity on any edge of G_h and d be the maximum node degree of H. We have the following theorem:

Theorem 15. *Algorithm AMHP can be implemented in* $O(n^2 \cdot m \cdot \log(n^2/m) \cdot \log(n \cdot c_{\max}) \cdot \log^2 n)$ *expected time or deterministically in* $O(n^3 \cdot m \cdot \log \cdot (n^2/m) \cdot \log(n \cdot c_{\max}) \cdot \log^2 n)$ *time. Furthermore, if the input hypergraph H has uniform weights on its nodes, AMHP can be implemented in* $O(n \cdot (d \cdot \log^2(m + \log n) + m \cdot \log n))$ *expected time.*

7. Conclusions

We have presented a multicommodity flow framework for deriving approximation algorithms for balanced bipartitioning and multiway partitioning problems for graphs and hypergraphs. The bounds are within polylogarithmic factors to the optimal. We then showed how to improve the time complexities of the described partitioning algorithms. Interesting open problems are: (a) to obtain tighter approximation bounds and (b) to reduce the time complexity.

One problem with many applications in VLSI is the *bisection* problem where the hypergraph is partitioned into two equal size parts so that the number of interconnecting hyperedges (or the sum of the weights on the interconnecting hyperedges) is minimized. In Ref. 27 we give approximation algorithms for the bisection problem on graphs and hypergraphs that satisfy certain constraints. Another interesting open problem is to obtain approximations for the bisection problem on any graph or hypergraph.

The presented approximation algorithms can be combined with existing heuristics to obtain partitions that maintain the approximation bound and are closer in practice to the optimal solution. It is an interesting question to determine experimentally whether provably good initial solutions for the heuristics in Refs. 11, 4, 9 trap us to local minima.

Acknowledgments

We acknowledge the anonymous referees for many useful suggestions.

References

1. T. Bui, C. Heigham, C. Jones and T. Leighton, "Improving the performance of the Kernighan-Lin and simulated annealing graph bisection algorithm", in *Proceedings of the* 25^{th} *ACM/IEEE Design Automation Conference*, 1988.
2. E. R. Barnes, A. Vannelli and J. Q. Walker, "A new heuristic for partitioning the nodes of a graph", *SIAM Journal of Discrete Mathematics*, vol.1, no.3, Aug. 1988, pp. 299–305.

3. C. K. Cheng and Y. C. A. Wei, "An improved two way partitioning algorithm with stable performance" *IEEE Transactions on CAD*, vol. 10, no 2, December 1991.
4. C. M. Fiduccia and R. M. Mattheyses, "A linear time heuristic for improving network partitions", in *Proceedings of the* 19^{th} *ACM/IEEE Design Automation Conference*, 1982, pp. 175–181.
5. L. R. Ford, Jr. and D. R. Fulkerson, *Flows in networks*. Princeton Univ. Press, Princeton, NJ., 1962. *Mathematics of Operations Research*, vol. 15, no. 3, 1990.
6. M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, San Francisco, 1979.
7. A. V. Goldberg and R. E. Tarjan, "Solving minimum cost flow problems by successive approximation", *Mathematics of Operations Research*, vol. 15, no. 3, 1990, pp. 430–466.
8. J. J. Hallenbeck, N. Kanopoulos and J. R. Cybrinski, "The Test Engineer's Assistant: A design environment for testable and diagnosable systems", *IEEE Transactions on Industrial Electronics*, special issue on electronic testing, May 1989.
9. D. S. Johnson, C. R. Aragon, L. A. McGeoch and C. Schevon, "Optimization by simulated annealing: An experimental evaluation (Part I)", *Preliminary draft, AT&T Bell Labs.*, Murray Hill, NJ, 1987.
10. M. Iri, "On an extension of the maximum-flow minimum cut theorem to multicommodity flows", *Journal of Operations Research Society of Japan*, vol. 13, no.3, Jan. 1971, pp. 129–135.
11. B. W. Kernighan and S. Lin, "An efficient procedure for partitioning graphs", *Bell Systems Technical Journal*, vol. 49, no. 2, February 1970.
12. P. Klein, A. Agrawal, R. Ramamurthy and S. Rao, "Approximation through multicommodity flow", in *Proceedings of the* 31^{st} *Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, 1990.
13. P. Klein, C. Stein and E. Tardos, "Leighton-Rao might be practical: faster approximation algorithms for concurrent flow with uniform capacities", in *Proceedings of the* 22^{st} *Annual ACM Symposium on Theory of Computing*, May 1990.
14. P. Klein, S. Plotkin, C. Stein and E. Tardos, "Faster approximation algorithms for the unit capacity concurrent flow problem with applications to routing and finding sparse cuts", *Technical report 961, School of Operations Research and Industrial Engineering, Cornell University*, 1991.
15. S. Kapoor and P. M. Vaidya, "Fast algorithms for convex quadratic programming and multicommodity flows", in *Proceedings of the 18th Annual ACM Symposium on Theory of Computing*, 1986, pp. 147–159.
16. T. Lengauer, *Combinatorial algorithms for integrated circuit layouts*. Teubner/Wiley series of applicable theory in computer science, N.Y., 1990.
17. T. Leighton, F. Makedon and S. Tragoudas, "Approximation algorithms for VLSI partitioning problems", in *Proceedings of the 1990 International Symposium on Circuits and Systems*, New Orleans, May 1–3, pp. 2865–2869.
18. T. Leighton, F. Makedon and S. Tragoudas, "Hypergraph partitioning algorithms", manuscript, *to be submitted.*
19. T. Leighton, F. Makedon, S. Plotkin, C. Stein, E. Tardos and S. Tragoudas, "Fast approximation algorithms for multicommodity flow problems", in *Proceedings of the* 23^{th} *Annual ACM Symposium on Theory of Computing*, New Orleans, May 1991, pp. 101–112.
20. T. Leighton, F. Makedon, S. Plotkin, C. Stein, E. Tardos and S. Tragoudas, "Fast approximation algorithms for multicommodity flow problems", *to appear in the Journal of Computer and System Sciences.*

21. T. Leighton and S. Rao, "An approximate max-flow min-cut theorem for uniform multicommodity flow problems with applications to approximation algorithms", in *Proceedings of the 29th Annual Symposium on Foundations of Computer Science*, Oct. 1988, pp. 422–431.
22. D. W. Matula and F. Shahroki, "Sparsest cuts and bottlenecks in graphs", *Discrete Applied Mathematics*, vol. 27, 1990, pp. 113–123.
23. F. Makedon and S. Tragoudas, "Approximating the minimum net expansion: Near optimal solutions to circuit partitioning problems", in *Proceedings of the 1990 Workshop on Graph Theoretic Concepts in Computer Science*, Berlin, Germany, Jun. 19–22.
24. S. Rao, "Finding near optimal separators in planar graphs", in *Proceedings of the* 28^{th} *Annual IEEE Symposium on Foundations of Computer Science*, 1987, pp. 225–237, a more detailed version appears as a Master's Thesis at the Laboratory for Computer Science, M.I.T.
25. Bryan Preas and M. Lorenzetti, *Physical Design Automation of VLSI Systems.* The Benjamin/Cummings Publishing Company Inc., 1988.
26. F. Shahroki and D. W. Matula, "The maximum concurrent flow problem", *Journal of the ACM*, vol. 37, 1990, pp. 318–334.
27. S. Tragoudas, "VLSI partitioning approximation algorithms using multicommodity flow and other techniques", *Ph.D. Thesis*, The University of Texas at Dallas, Richardson, Texas 75083-0688, Jun. 1991.
28. S. Tragoudas, R. Farrell and F. Makedon, "Circuit partitioning into small sets: a tool to support testing with further applications", in *Proceedings of the* 2^{nd} *European Design Automation Conference*, Amsterdam, The Netherlands, 25–28 Feb. 1991, pp. 518–523. To appear in the Journal of Microprocessors and Microsystems.
29. P. M. Vaidya, "Speeding up linear programming using fast matrix multiplication", in *Proceedings of the* 30^{th} *Annual IEEE Symposium on Foundations of Computer Science*, 1989 pp. 332–337.

Algorithmic Aspects of VLSI Layout
Eds. M. Sarrafzadeh and D. T. Lee
© World Scientific Publishing Co.

INTEGER PROGRAM FORMULATIONS OF GLOBAL ROUTING AND PLACEMENT PROBLEMS

T. LENGAUER* and M. LÜGERING†
University of Paderborn
33095 Paderborn, Germany

ABSTRACT

Global routing is an essential phase during the process of physical design of integrated circuits. Combinatorially, this problem amounts to a set of interdependent Steiner tree problems. Several versions of the problem are of importance in practical applications. All of them can be formulated as integer programs. Several such formulations have been investigated in the past, and different solution methods have been developed for different formulations.

In this paper we give an overview of integer program formulations of the global routing problem and their solution methods, and we introduce new concepts for solving this important combinatorial problem. Finally, we present integer program formulations that integrate placement with global routing.

Keywords: Global routing, placement, Steiner trees, integer programming, Lagrange relaxation, circuit layout, facets.

1. Introduction

Global routing is an essential part of circuit layout. The global routing phase usually follows the placement or floorplanning phase. During the global routing phase, the approximate course of all wires is determined. A detailed introduction into the global routing problem and its role during circuit layout can be found in Ref. 1.

Combinatorially, an instance of the global routing problem has the following elements:

A routing graph: This undirected graph is denoted by $G = (V, E)$. The graph G is the representation of the routing regions on the chip that result from the preceding placement/floorplanning phase. In many applications, G is planar.

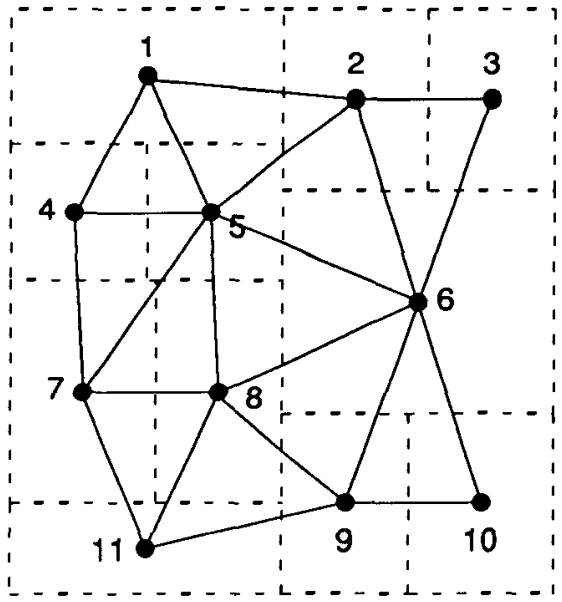
*Current address: *German National Computer Science Research Center (GMD-I1), Schloß Birlinghoven, 53757 St. Augustin 1, Germany. Email: lengauer@gmd.de* and *Department of Computer Science, University of Bonn, Römerstraße 164, 53117 Bonn, Germany*
†Current address: *Department of Computer Science, University of Bonn, Römerstraße 164, 53117 Bonn, Germany. Email: ml@informatik.uni-bonn.de*

In general, this does not have to be the case, however. Intuitively, the vertices in the routing graph represent possible positions of wire terminals, and the edges represent channels along which wiring can be performed. In order to support this intuition, each edge e in the routing graph has two labels, its *length* $\ell(e) \geq 0$ and its *capacity* $c(e) \geq 0$. We assume capacities to be integer. Depending on the wiring model, the interpretation of G can take on different forms. Figures 1(a) and (b) depict two different wiring models. In Fig. 1(a), the vertices are located at centers of the cells of the floorplan. The edges represent adjacencies between cells in the floorplan. In this model, a reasonable choice of the labels for an edge $e = \{v, w\}$ is to choose $\ell(e)$ to be the actual length of e in the planar embedding shown in the figure — e.g., according to the Manhattan distance or the Euclidean distance — and $c(e)$ to be the length of the *dual* edge e' of e in the floorplan, in an appropriate unit. In this way, the capacity $c(e)$ measures the number of wires that can cross e' when moving from cell v to cell w.

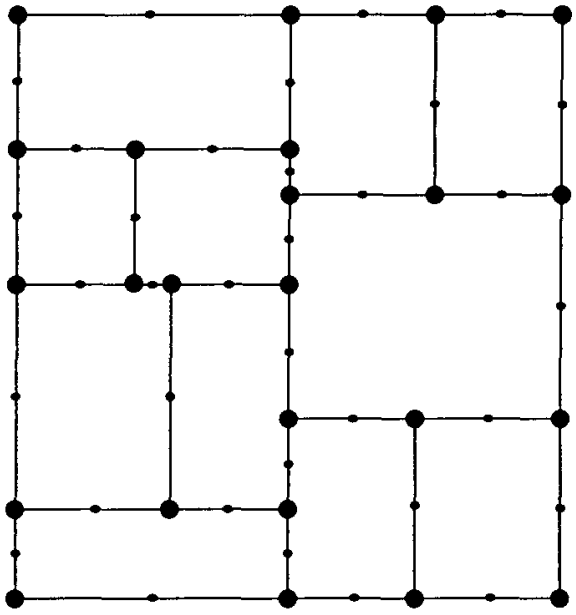
In Fig. 1(b), G represents the channel structure of the floorplan. In this case, the length of an edge should again be its actual length in the embedding. Its capacity should be an estimate of the width of the corresponding channel.

A set of nets: This multiset is denoted by $N \subseteq 2^V$. Each net is given by the set of its terminals, and each terminal is a vertex in V. Thus, in Fig. 1(a), all terminals for a cell are clustered in the center of this cell. In Fig. 1(b), vertices can be added as desired to represent specific terminal positions. Of course, the same net can occur multiply, with each instance of the net being routed differently. Therefore, N is a multiset, in general. We denote a net with ν, and different copies of ν with (ν, i), $i = 1, \ldots, k_\nu$. (k_ν is the *multiplicity* of net ν.) Furthermore, each net (ν, i) has an integer *weight* $w(\nu, i) > 0$ that represents the cost of a unit-length wire for net (ν, i). Often, all weights will be unity. Different weights can, for instance, model the different bit-width of busses that are represented by a single net each.

A set of admissible routes for each net: Often, all Steiner trees between the vertices in ν will be admissible. However, we also allow to restrict ourselves to specific routes, such as all routes of a given maximal length or well-shaped routes, say, with few bends. The admissible routes for net (ν, i) will be denoted with $T^1_{\nu,i}, \ldots, T^{I_{\nu,i}}_{\nu,i}$. The set of all admissible routes is denoted by $\mathcal{T} = (T^j_{\nu,i})_{1 \leq j \leq I_{\nu,i}, (\nu,i) \in N}$. Usually, the number $I_{\nu,i}$ of admissible routes for (ν, i) is large (exponential in the number of terminals of ν). But often, the set of admissible routes can be defined concisely in small space. (For instance, often routes with no more than b bends are considered, where b is a small constant. In this case, the admissible routes can be specified just by including the constant b in the input. Given this information, the admissible routes themselves can be generated by the routing algorithm.)



(a)



(b)

Fig. 1. Two different routing models. (a) Channelless routing. (b) Routing in channels.

There are a multitude of variants of the global routing problem. In general, they fall into two classes, *constrained global routing* and *unconstrained global routing.* In the constrained global routing problem, the edge capacities in G are strictly adhered to — violations lead to illegal routings. In the unconstrained version, routings exceeding the edge capacities are allowed but punished in the cost function. The

total wire length enters the cost measure with low priority, in both versions of the global routing problem.

We now define the solutions of these two versions of the global routing problem formally. In order to do so, we need to define a few notions. For this purpose, let $I = (G, N, \mathcal{T})$ be the given problem instance.

Routing: A *routing* $\mathcal{R} = (T_{\nu,i})_{(\nu,i)\in N'}$ is a set of admissible routes $T_{\nu,i}$ for a subset N' of the nets N. The nets in $N \setminus N'$ have no routes in $\mathcal{R}$. If $N \setminus N' \neq \emptyset$, the routing $\mathcal{R}$ is said to be *incomplete*.

Traffic: The *traffic* $U(\mathcal{R}, e)$ across edge $e \in E$ in a routing $\mathcal{R}$ is the total weighted cost of all nets that are wired across e in $\mathcal{R}$, i.e.,

$$U(\mathcal{R}, e) := \sum_{\substack{(\nu,i)\in N' \\ e\in T_{\nu,i}}} w(\nu, i)$$

Load: The *load* of edge $e \in E$ in a routing $\mathcal{R}$ is defined to be

$$\Lambda(\mathcal{R}, e) := U(\mathcal{R}, e) - c(e)$$

If $\Lambda(e) < 0$ then the edge is said to be *unsaturated*, if $\Lambda(e) = 0$, then e is *saturated* and if $\Lambda(e) > 0$ then e is *oversaturated*. (In Ref. 1, the load is defined as $\Lambda(\mathcal{R}, e) := U(\mathcal{R}, e)/c(e)$. The difference between these two definitions is technical, but the definition given here is more attractive in many practical settings, because channels with many free tracks are avoided.[2])

Constrained Global Routing: A *legal routing* w.r.t. the constrained global routing problem is a routing $\mathcal{R}$ such that no edge $e \in E$ is oversaturated. The cost of the routing $\mathcal{R}$ is the pair

$$\left(\sum_{(\nu,i)\in N\setminus N'} w(\nu, i), W(\mathcal{R}) \right)$$

where

$$W(\mathcal{R}) := \sum_{e\in E} \ell(e) U(\mathcal{R}, e)$$

is the total weighted wire length of the routing. Costs of different routings are compared using the lexicographic ordering. Thus, the number of routed nets is maximized with first priority. Among all routings that route a maximum number of nets, the one with the smallest total weighted wire length is chosen.

Unconstrained Global Routing: W.r.t. the unconstrained global routing problem, each complete routing is legal. The cost of the routing is the pair

$$\left(\max_{e\in E} \Lambda(\mathcal{R}, e), W(\mathcal{R}) \right)$$

Costs of different routings are again compared using the lexicographic ordering. Thus, an optimal routing is one with minimum maximal edge load and, among those, one with minimum total weighted wire length.

Both versions of the global routing problem are strongly NP-hard. In fact, this holds even for severely restricted cases of these problems:

- If $|N| = 1$, we obtain the minimum Steiner tree problem, which is strongly NP-hard.[3]
- Kramer and van Leeuwen[4] have shown the restriction to be strongly NP-hard, in which all nets have exactly two terminals, all capacities are unity, all edge lengths are zero, and the routing graph is a square grid graph. (The restriction of the constrained global routing problem, in which all capacities are unity and all edge lengths are zero is also called *Steiner tree packing.*)
- Korte *et al.*[5] have proved the Steiner tree packing problem to be strongly NP-hard even if $|N| = 2$ and G is planar.

Beside these complexity issues, both versions of the global routing problem have received quite some attention from the CAD community, in the past few years. In the following, we sketch two advanced methods of global routing. For details, see Ref. 1.

The method of *hierarchical global routing* solves the constrained global routing problem. This method has been introduced by Burstein and Pelavin[6] in the context of gate array layout and subsequently extended to general floorplans by Luk *et al.*[7] Burstein and Hong[8] integrated this approach to global routing with placement in the context of gate-array layout, and Lengauer and Müller[9] did so for general floorplans. Hierarchical global routing methods use a cut tree for the floorplan, such as is obtained by floorplanning methods based on circuit partitioning. A small global routing problem is associated with each node in the cut tree and solved exactly, using integer programming methods. The partial solutions thus obtained are put together to form a "good" solution of the whole problem instance. Whereas the solutions of the small global routing problems pertaining to the nodes in the cut tree are optimal, no statement about the quality of the resulting overall solutions can be made. However, experience indicates that the solutions are quite good, in practice.

The method of *routing by linear relaxation* considers the unconstrained global routing problem. In fact, it considers the special case that disregards edge lengths ($\ell(e) = 0$ for all edges e in the routing graph). This technique formulates the global routing problem as an integer program and first solves a linear relaxation of this program. Then it uses the fractional outcomes as biases for appropriate coin tosses that generate a solution in a randomized fashion. Repeating this random experiment an appropriate number of times generates a good solution with high probability. A worst case analysis of this method has been given in Ref. 10. While

this analysis is extraordinarily tight in the asymptotic sense, it does not provide tight bounds for the range of costs actually provided by routing problems. For typical values, the analysis promises to be within between about 50% and 100% above the optimum value. However, it has been reported that this method is much more effective on small gate arrays.[11] In fact, it often comes up with an optimum answer. The method has yet to be adapted to large gate arrays. A deterministic version of the method has been presented in Ref. 12.

In general, we can make the following remarks in terms of a comparison of the two versions of the global routing problem. Constrained global routing is the harder problem in so far, as it is not sufficient to come up with just any routing; it has to be a *legal* one. Ensuring legality of the routings is quite a difficult issue, in general. So far, this problem has stood in the way of the use of randomized algorithms based on the linear relaxation of constrainted global routing integer programs, for instance. In general, if we are looking for heuristic methods for finding good routings, it is very comforting to not have to deal with legality issues. Furthermore, most often the capacities that are attached to the edges of a routing graph are only rough estimates, anyway, that arise during floorplanning. Thus there is no need to adhere strictly to these capacities.

In our opinion, there are only two reasons for considering the constrained global routing problem. First, there may be applications, in which the edge capacities are indeed tight constraints. For instance, on gate arrays channels are prefabricated with fixed capacities. Thus, we can solve the constrained global routing problem. In general, we will get an incomplete solution that we have to complete, for instance, by heuristic rip-up and reroute techniques. However, even here we can solve the unconstrained global routing problem, instead. If we turn up with a maximum edge load of at most 0 the solution is fabricable. Otherwise, we know that there is no feasible solution and, what is more, we are given a solution with the fewest violations, in some sense.

The other reason why the constrained global routing problem is interesting is that it is a generalization of the routing problem in the knock-knee model, which it reduces to by setting all edge capacities to unity. There is a lot of research on knock-knee routing,[1] leading to a large potential for cross-fertilization between these two problem areas. However, even unconstrained global routing shows many affinities to knock-knee routing. So, in our opinion, the unconstrained global routing problem is the more attractive version of the global routing problem.

Both of the routing methods described above rely on integer program formulations of the global routing problem. In fact, integer program formulations are quite useful above and beyond the two methods just presented: They make available to us a whole host of methods of discrete optimization. The purpose of this paper is to investigate global routing as an integer program, to introduce different integer program formulations of global routing, and to discuss their respective merits. We will also point out concepts for new optimization methods for the global routing problem.

Section 2 will introduce and motivate different integer program formulations of the two versions of global routing. Section 3 will present the polyhedra of these integer programs and discuss structural characteristics, which are relevant to optimization aspects of the respective formulations. Section 4 will propose new solution methods for solving the global routing problem. In Sec. 5 we discuss integer program formulations for integrating placement with global routing. Finally, Sec. 6 will give a summary.

2. Integer Program Formulations of Global Routing Problems

In this section we discuss several integer program (IP) formulations of both versions of the global routing problem. We will introduce the programs formally and discuss their merits.

The global routing problem has two sources of combinatorial complexity. First, the minimum Steiner tree problem is part of global routing, and this problem is hard, in itself. Second, the interdependence between different nets introduces additional complexity. In our IP formulations we will sometimes aim at separating these two sources of complexity. Doing so, will enable us to apply decompositions that, in the end, will ease the task of global routing.

2.1. The Explicit Approach

The most straightforward way of dealing with the minimum Steiner tree problem is to enumerate all possible Steiner trees, rather than letting the integer program find Steiner trees as part of the optimization. This observation leads to the so-called *explicit approach* to global routing IPs. In this approach, we provide a separate binary variable $x_{\nu,i,j}$ for each net $(\nu,i) \in N$ and each admissible route $T^j_{\nu,i}$ of net (ν,i). We interpret $x_{\nu,i,j} = 1$ to mean that net (ν,i) takes route $T^j_{\nu,i}$, and $x_{\nu,i,j} = 0$ to mean the opposite fact. The constraints of the IP have to enforce this interpretation. Therefore, we impose the following *completeness constraints*:

$$\sum_{j=1}^{I_{\nu,i}} x_{\nu,i,j} = 1 \quad \text{for all } (\nu,i) \in N$$

In constrained global routing, we allow for incomplete routings. In the IP formulation we can do so by providing, for each net, a special admissible route, the so-called *empty route*, which does not use any edge. We assume this route to have the index $j = 0$, for each net. In this case, the completeness constraints have to sum from $j = 0$ to $j = I_{\nu,i}$. In unconstrained global routing, empty routes are forbidden.

The remainder of the IP differs between the constrained and the unconstrained version of the global routing problem. However, in both cases we use the following linear functions of the variables of the IP:

Traffic:

$$U(x,e) := \sum_{\substack{(\nu,i)\in N \\ 1\le j\le I_{\nu,i} \\ e\in T^j_{\nu,i}}} w(\nu,i)\, x_{\nu,i,j}$$

Load:

$$\Lambda(x,e) := U(x,e) - c(e)$$

Total Weighted Wire Length:

$$W(x) := \sum_{e\in E} \ell(e)\, U(x,e)$$

Constrained Global Routing: We have to impose the following *capacity constraints*:

$$\Lambda(x,e) \le 0 \quad \text{for all } e \in E$$

The cost function of the IP is

$$c_{\mathrm{CGR}}(x) := Z \cdot \sum_{(\nu,i)\in N} w(\nu,i) x_{\nu,i,0} \;+\; W(x)$$

The value Z is chosen large enough such that the first term dominates the second and the lexicographic ordering is imposed. Choosing $Z = 1 + \sum_{(\nu,i)\in N} w(\nu,i) \cdot \sum_{e\in E} \ell(e)$ is sufficient.

The resulting IP is denoted by $I^{\mathrm{expl}}_{\mathrm{CGR}}$.

Unconstrained Global Routing: We provide an additional *load variable* x_L, and replace the capacity constraints by the following *load constraints*:

$$\Lambda(x,e) - x_L \le 0 \quad \text{for all } e \in E$$

Note that x_L can be negative. If x_L were to become positive, for some reason, we can replace the right hand side of each load constraint with a suitably small fixed constant b, thus translating x_L into the positive reals. ($b = -\max\{c(e) \mid e \in E\}$ is an appropriate choice.)

The cost function of the IP is

$$c_{\mathrm{UGR}}(x) := Z \cdot x_L \;+\; W(x)$$

Choosing Z at least as large as above again imposes the desired lexicographic ordering. (If $\ell = 0$ for all $e \in E$, then the cost function can be reduced to $c_{\mathrm{UGR}}(x) = x_L$.)

The resulting IP is denoted by $I^{\mathrm{expl}}_{\mathrm{UGR}}$.

The explicit approach has the advantage that it trivializes the minimum Steiner tree optimization. In fact, since each route receives a private variable, the minimization for a single net just amounts to selecting the appropriate variable.

This feature is obtained at a high cost. Indeed, there is an exponential number of variables, in general. While this fact seems to render the explicit approach infeasible, at first sight, we will see that this problem can be overcome, by decomposing the IP in such a way that an explicit handling becomes unecessary for many or all variables.

A second advantage of the explicit approach is that it enables us to restrict ourselves to *any* subset of admissible routes: We just provide variables only for the admissible routes. If the set of admissible routes is not too large, then the IP can be generated explicitly. Ng *et al.*[11] have successfully used this feature in practical experiments with global routing. Furthermore, this observation enables us to incorporate additional side constraints into global routing. For instance, in timing-driven layout, we often are only interested in wires whose length lies between specified lower and upper bounds.[13] In the explicit approach, we just render routings inadmissible that violate these constraints.

As a general rule we can state: Whenever the number of routes is small or we are not concerned with Steiner tree computations, we can use the explicit approach.

2.2. The Implicit Approach

The essential disadvantage of the explicit approach is the exploding number of variables of the IP. This phenomenon can be dealt with by not providing an explicit variable for each route but implicitly representing all routes. We call this approach the *implicit approach*. Let us assume, for now, that all Steiner trees are admissible for each net.

We provide a binary variable $x_{\nu,i,e}$ for each net $(\nu, i) \in N$ and each edge $e \in E$. The interpretation is that, if $x_{\nu,i,e} = 1$, then the routing for net (ν, i) takes edge e, otherwise it does not. Now, the number of variables has been reduced significantly. In fact, there are only $|N||E|$ variables.

The completeness constraints now have to be replaced by the so-called *Steiner cut constraints*. These constraints ensure that the assignments to the variables of the IP represent Steiner trees. The constraints look as follows.

Let $(\nu, i) \in N$ be a net. We call a partition $(X, V \setminus X)$ of the vertices of G into two disjoint parts such that each of the parts contain at least one terminal of (ν, i), i.e., $X \cap (\nu, i), (V \setminus X) \cap (\nu, i) \neq \emptyset$, a *Steiner cut* for net (ν, i). The Steiner cut constraints are

$$\sum_{\substack{e=(v,w)\in E \\ v\in X, w\in V\setminus X}} x_{\nu,i,e} \geq 1 \quad \text{for } (\nu, i) \in N,\ (X, V \setminus X) \text{ Steiner cut of } (\nu, i)$$

The set of Steiner cuts for (ν, i) grows exponentially with the number of terminals of (ν, i) and with the number of vertices in G. Thus integer programs for the implicit

approach have few variables but a huge number of constraints. However, there are methods for solving integer programs without generating all constraints. Therefore, we will not be worried by the large number of constraints, for now.

A more critical problem poses the fact, that the implicit formulation is not as robust with respect to additional side constraints as the explicit formulation. Restricting or attention to routes with few bends or a specific length is more difficult, in this approach. We will come back to this problem in Sec. 4.6.

Let us now complete the IP for the implicit approach in the constrained and unconstrained version of the global routing problem. First, we formulate the traffic in terms of the new variables.

Traffic:

$$U(x,e) := \sum_{(\nu,i)\in N} w(\nu,i)\, x_{\nu,i,e}$$

The load and total weighted wire length are formulated in terms of the traffic in the same way as before.

Constrained Global Routing: The capacity constraints are formulated in terms of the traffic as in the explicit approach.

The formulation of the Steiner cut constraints allows only for *complete* routings. Since we want to deal with partial routings, as well, we provide, for each net (ν,i), a variable $x_{\nu,i,0}$ representing the empty route for the net. The Steiner cut constraints for net (ν,i) then look as follows:

$$x_{\nu,i,0} + \sum_{\substack{e=(v,w)\in E\\ v\in X, w\in V\setminus X}} x_{\nu,i,e} \geq 1 \quad \text{for } (X, V\setminus X) \text{ Steiner cut of } (\nu,i)$$

With this provision, the cost function is chosen as in the explicit approach.

The resulting IP is denoted by $I^{\text{impl}}_{\text{CGR}}$.

Unconstrained Global Routing: As in the explicit approach, we provide an additional *load variable* x_L, and, in addition to the Steiner cut constraints, impose the *load constraints*:

$$\Lambda(x,e) - x_L \leq 0 \quad \text{for all } e \in E$$

The cost function of the IP is

$$c_{\text{UGR}}(x) := Z \cdot x_L \; + \; W(x)$$

where Z is chosen as before.

The resulting IP is denoted by $I^{\text{impl}}_{\text{UGR}}$.

In the following, we assume that $G = (V, E)$ is a biconnected graph. This does not restrict the generality because, we can route on each biconnected component of a graph separately, and afterwards combine the solutions. The assumption, that G is biconnected simplifies the structure of the polyhedra of global routing problems.

3. The Polyhedra for Global Routing Integer Programs

In this section we discuss the polyhedra related to the integer programs of the last section. We suppose a basic familiarity with polyhedral theory. Readers can familiarize themselves with these subjects using Refs. 14, 15, 16, 17. Again, we distinguish between the explicit and the implicit approach.

For instances of the constrained global routing problem, we can assume that every route can be used for the routing, e.g., no route exceeds any of the edge capacities. If we use the explicit approach, this assumption does not restrict the generality, because there is no sense in making the routes excluded in this way admissible. In the implicit approach, the restriction may be real, but the problem does not occur often, in practice.

3.1. The Explicit Approach

First we define formally the polytopes of the constrained global routing problem and the unconstrained global routing problem, respectively, while using the explicit approach.

For our further discussion let ξ be the total number of feasible routes for all nets, i.e.,

$$\xi := \sum_{(\nu,i)\in N} (I_{\nu,i} + 1)$$

in the case of constrained global routing, and

$$\xi := \sum_{(\nu,i)\in N} I_{\nu,i}$$

in the case of unconstrained global routing.

3.1.1. Definitions

Constrained Global Routing: Let $P_{\mathrm{CGR}}^{\mathrm{expl}}$ be the polytope of the integer program $I_{\mathrm{CGR}}^{\mathrm{expl}}$. More precisely,

$P_{\mathrm{CGR}}^{\mathrm{expl}} := \mathrm{conv}\{x \in \mathbb{R}^{\xi} \mid x$ is the concatenation of $|N|$ vectors $x_{\nu,i} \in \mathbb{R}^{I_{\nu,i}+1}$ (one vector for each net), such that, for all $(\nu, i) \in N$, $\sum_{j=0}^{I_{\nu,i}} x_{\nu,i,j} = 1$, and $x_{\nu,i,j} \in \{0,1\}$ for all $j = 0, \ldots, I_{\nu,i}$, and $U(x,e) \leq c(e)$ for all $e \in E\}$

Let $P_{\mathrm{CGR}}^{\mathrm{expl,R}}$ be the polytope of the integer program $I_{\mathrm{CGR}}^{\mathrm{expl,R}}$ which is obtained from $I_{\mathrm{CGR}}^{\mathrm{expl}}$ by disregarding the capacity constraints. (The R stands for *relaxed*, since we omit the computationally difficult capacity constraints.)

$P_{\mathrm{CGR}}^{\mathrm{expl,R}} := \mathrm{conv}\{x \in \mathbb{R}^{\xi} \mid x$ is the concatenation of $|N|$ vectors $x_{\nu,i} \in \mathbb{R}^{I_{\nu,i}+1}$ (one vector for each net), such that, for all $(\nu,i) \in N$, $\sum_{j=0}^{I_{\nu,i}} x_{\nu,i,j} = 1$, and $x_{\nu,i,j} \in \{0,1\}$ for all $j = 0,\ldots,I_{\nu,i}\}$

Unconstrained Global Routing: Let $P_{\mathrm{UGR}}^{\mathrm{expl}}$ be the polytope of the integer program $I_{\mathrm{UGR}}^{\mathrm{expl}}$. More precisely,

$P_{\mathrm{UGR}}^{\mathrm{expl}} := \mathrm{conv}\{x \in \mathbb{R}^{\xi+1} \mid x$ is the concatenation of a variable x_L with $|N|$ vectors $x_{\nu,i} \in \mathbb{R}^{I_{\nu,i}}$ (one vector for each net), such that, for all $(\nu,i) \in N$, $\sum_{j=1}^{I_{\nu,i}} x_{\nu,i,j} = 1$ and $x_{\nu,i,j} \in \{0,1\}$ for all $j = 1,\ldots,I_{\nu,i}$, and $U(x,e) - c(e) \leq x_L$ for all $e \in E\}$

Let $P_{\mathrm{UGR}}^{\mathrm{expl,R}}$ be the polyhedron of the integer program $I_{\mathrm{UGR}}^{\mathrm{expl,R}}$ which is obtained from $I_{\mathrm{UGR}}^{\mathrm{expl}}$ by disregarding the load constraints.

$P_{\mathrm{UGR}}^{\mathrm{expl,R}} := \mathrm{conv}\{x \in \mathbb{R}^{\xi+1} \mid x$ is the concatenation of a variable x_L with $|N|$ vectors $x_{\nu,i} \in \mathbb{R}^{I_{\nu,i}}$ (one vector for each net), such that, for all $(\nu,i) \in N$, $\sum_{j=1}^{I_{\nu,i}} x_{\nu,i,j} = 1$, and $x_{\nu,i,j} \in \{0,1\}$ for all $j = 1,\ldots,I_{\nu,i}\}$

Whenever we need to ensure the boundedness of the polyhedra, we implicitly assume that $b \leq x_L \leq B$ where, b and B are chosen appropriately, such as not to restrict the optimization. $B = 1 + \sum_{(\nu,i)\in N} w(\nu,i)$ and $b = -\max\{c(e) \mid e \in E\}$ are sufficient for this purpose.

3.1.2. Facets

We now give some intuition on the shape of the polytopes.

The polytope $P_{\mathrm{CGR}}^{\mathrm{expl,R}}$ is a $\xi - |N|$-dimensional polytope in ξ-space whose facets are the *trivial facets* defined by the inequalities $x_{\nu,i,j} \geq 0$, $(\nu,i) \in N$, $j = 0,\ldots,I_{\nu,i}$, and which lies in the affine subspace fulfilling all equations $\sum_{j=0}^{I_{\nu,i}} x_{\nu,i,j} = 1$ for $(\nu,i) \in N$. This polyhedron has only integral corners, since the constraint matrix of the underlying integer program $I_{\mathrm{CGR}}^{\mathrm{expl,R}}$ is totally unimodular.

The polyhedron $P_{\mathrm{UGR}}^{\mathrm{expl,R}}$ is obtained by lifting $P_{\mathrm{CGR}}^{\mathrm{expl,R}}$ along the new dimension x_L. (Here we disregard the empty routes, however.) $P_{\mathrm{UGR}}^{\mathrm{expl,R}}$ is a $\xi - |N| + 1$-dimensional polyhedron in $\xi + 1$-space.

Adding capacity constraints in the case of constrained global routing and load constraints in the case of unconstrained global routing cuts off corners from the polyhedra $P_{\mathrm{CGR}}^{\mathrm{expl,R}}$ and $P_{\mathrm{UGR}}^{\mathrm{expl,R}}$, respectively. While these cuts do not decrease the dimension of the involved polyhedra, they introduce new corners that may be non-integral. The facet-inducing cuts among these are called *capacity facets* and *load facets*, respectively, the resulting polyhedra are denoted by $P_{\mathrm{CGR}}^{\mathrm{expl,S}}$ and $P_{\mathrm{UGR}}^{\mathrm{expl,S}}$, respectively. (The S stands for *sliced* since we slice off parts from $P_{\mathrm{CGR}}^{\mathrm{expl,R}}$ and $P_{\mathrm{UGR}}^{\mathrm{expl,R}}$, respectively, to get these polyhedra.)

The polyhedra $P^{\text{expl}}_{\text{CGR}}$ and $P^{\text{expl}}_{\text{UGR}}$ are the convex hulls of all integral points in the polyhedra $P^{\text{expl,S}}_{\text{CGR}}$ and $P^{\text{expl,S}}_{\text{UGR}}$, respectively. The corners of $P^{\text{expl}}_{\text{CGR}}$ and $P^{\text{expl}}_{\text{UGR}}$ are therefore exposed by cutting the polyhedra $P^{\text{expl,S}}_{\text{CGR}}$ and $P^{\text{expl,S}}_{\text{UGR}}$ with additional facet-defining hyperplanes. Let us call these hyperplanes *I-facets*, since they encapsulate the interdependence between different nets in global routing.

In order to illustrate this somewhat involved situation consider the example depicted in Fig. 2. We set $\ell, c, w \equiv 1$. There are two nets with two terminals each. For every net ν_i, $i = 1,2$ there are two feasible routes $T^1_{\nu_i}$, and $T^2_{\nu_i}$.

The corresponding polyhedra $P^{\text{expl,R}}_{\text{UGR}}$, $P^{\text{expl,S}}_{\text{UGR}}$ $P^{\text{expl}}_{\text{UGR}}$ are three-dimensional and reside in 5-space. More precisely, they reside in a three-dimensional subspace of 5-space that is defined by the equations $x_{\nu_i,1} + x_{\nu_i,2} = 1$ for both nets ν_i, $i = 1,2$. Inside this space, the polyhedra look as shown in Fig. 3. The figure reveals that there are four load facets but there is only one I-facet, namely the facet $x_L \geq 1$. Finding this facet is tantamount to solving the global routing problem itself.

The same problem instance for constrained global routing shares the same difficulty. One can easily verify that the inequality $x_{\nu_1,0} + x_{\nu_2,0} \geq 1$ defines a facet of the polyhedron $P^{\text{expl}}_{\text{CGR}}$. The term $x_{\nu_1,0} + x_{\nu_2,0}$ is the leading term in the cost function of this example. Thus finding this facet is tantamount to solving the problem itself.

We conjecture that this is not an isolated example but that, in many cases, there are I-facets of the form $x_L \geq c$, where c is the optimal cost for the routing. This could mean that it is unlikely that cutting-plane algorithms are a feasible tool for solving global routing problems.

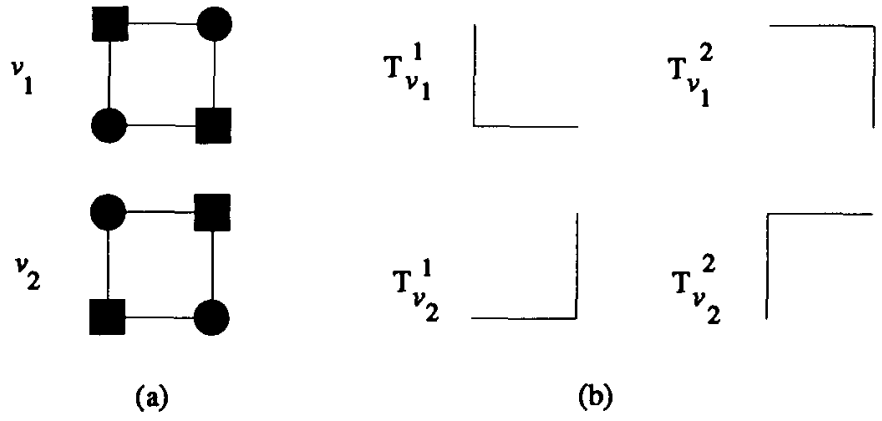


Fig. 2. A small routing example. (a) The two nets, terminal are denoted by squares. (b) The feasible routes.

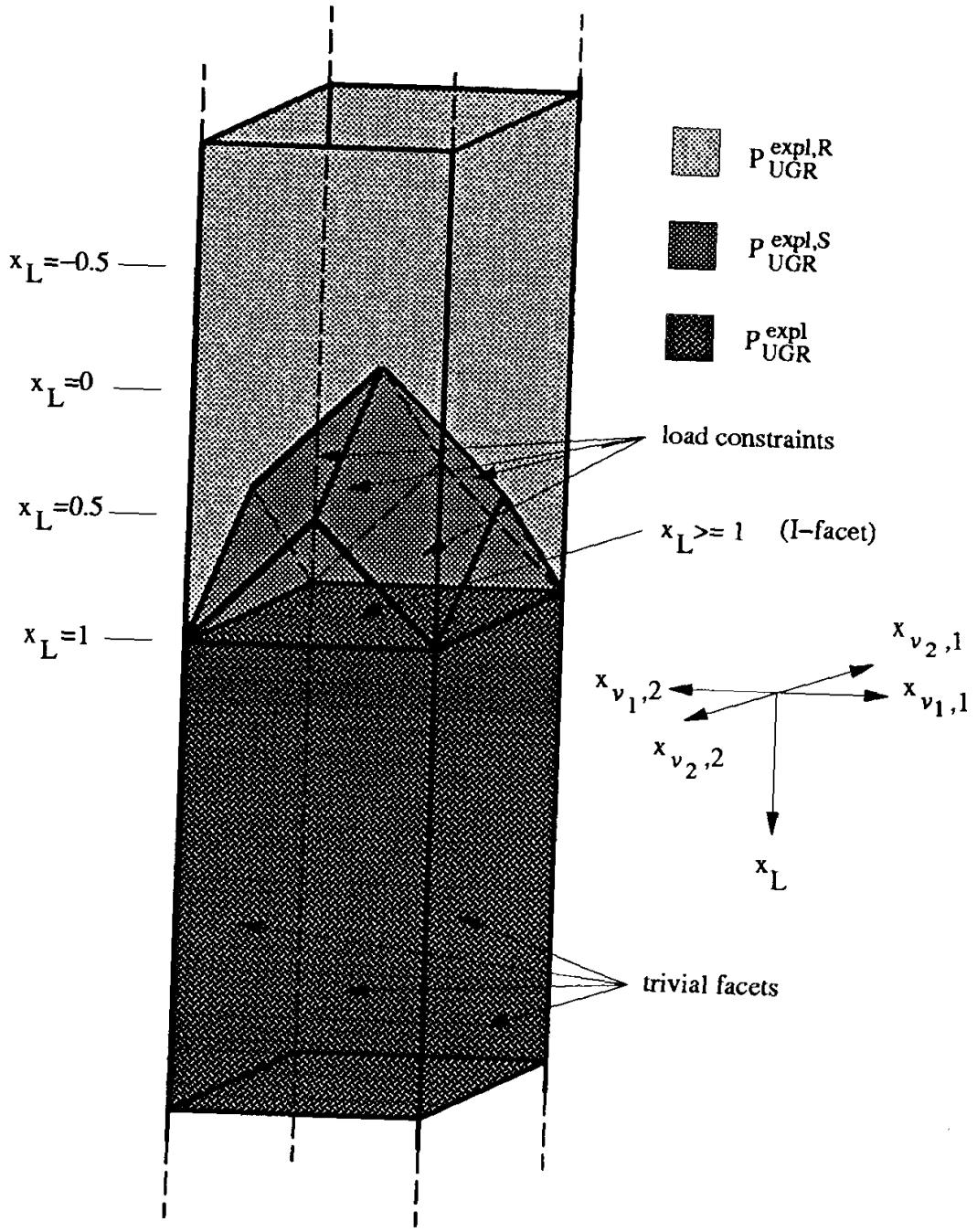


Fig. 3. The polyhedra of the example from Fig. 2.

3.2. The Implicit Approach

Let us now define formally the polyhedra of the constrained global routing problem and the unconstrained global routing problem, respectively, while using the implicit approach.

3.2.1. Definitions

Constrained Global Routing: Let $P_{\mathrm{CGR}}^{\mathrm{impl}}$ be the polytope of the integer program $I_{\mathrm{CGR}}^{\mathrm{impl}}$. More precisely,

$P_{\mathrm{CGR}}^{\mathrm{impl}} := \mathrm{conv}\{x \in \mathbb{R}^{|N|(|E|+1)} \mid x$ is the concatenation of $|N|$ vectors $(x_{\nu,i}, x_{\nu,i,0}) \in \mathbb{R}^{|E|+1}$ (one vector for each net), such that, for all $(\nu,i) \in N$, either $x_{\nu,i,e}$, $e \in E$, meets all Steiner cut inequalities and $x_{\nu,i,0} = 0$, or else $x_{\nu,i,0} = 1$ and $x_{\nu,i,e} \in \{0,1\}$ for all $e \in E$. Furthermore $U(x,e) - c(e) \leq 0$ for all $e \in E\}$.

Let $P_{\mathrm{CGR}}^{\mathrm{impl,R}}$ be the polytope of the integer program $I_{\mathrm{CGR}}^{\mathrm{impl,R}}$ which is obtained from $I_{\mathrm{CGR}}^{\mathrm{impl}}$ by disregarding the capacity constraints.

$P_{\mathrm{CGR}}^{\mathrm{impl,R}} := \mathrm{conv}\{x \in \mathbb{R}^{|N|(|E|+1)} \mid x$ is the concatenation of $|N|$ vectors $(x_{\nu,i}, x_{\nu,i,0}) \in \mathbb{R}^{|E|+1}$ (one vector for each net), such that, for all $(\nu,i) \in N$, either $x_{\nu,i,e}$, $e \in E$, meets all Steiner cut inequalities and $x_{\nu,i,0} = 0$, or else $x_{\nu,i,0} = 1$, and $x_{\nu,i,e} \in \{0,1\}$ for all $e \in E\}$.

Unconstrained Global Routing: Let $P_{\mathrm{UGR}}^{\mathrm{impl}}$ be the polyhedron of the integer program $I_{\mathrm{UGR}}^{\mathrm{impl}}$. More precisely,

$P_{\mathrm{UGR}}^{\mathrm{impl}} := \mathrm{conv}\{x \in \mathbb{R}^{|N||E|+1} \mid x$ is the concatenation of a variable x_L with $|N|$ vectors $x_{\nu,i} \in \mathbb{R}^{|E|}$ (one vector for each net), such that, for all $(\nu,i) \in N$, the $x_{\nu,i,e}$, $e \in E$ meet all Steiner cut inequalities, and $x_{\nu,i,e} \in \{0,1\}$ for all $e \in E$, and $U(x,e) - c(e) \leq x_L$ for all $e \in E\}$.

Let $P_{\mathrm{UGR}}^{\mathrm{impl,R}}$ be the polyhedron of the integer program $I_{\mathrm{UGR}}^{\mathrm{impl,R}}$ which is obtained from $I_{\mathrm{UGR}}^{\mathrm{impl}}$ by disregarding the load constraints.

$P_{\mathrm{UGR}}^{\mathrm{impl,R}} := \mathrm{conv}\{x \in \mathbb{R}^{|N||E|+1} \mid x$ is the concatenation of a variable x_L with $|N|$ vectors $x_{\nu,i} \in \mathbb{R}^{|E|}$ (one vector for each net), such that, for all $(\nu,i) \in N$, the $x_{\nu,i,e}$, $e \in E$ meet all Steiner cut inequalities, and $x_{\nu,i,e} \in \{0,1\}$ for all $e \in E\}$.

If we need to assert the boundedness of all polyhedra, we again implicitly assume that $b \leq x_L \leq B$ is bounded by suitable constants b and B. Again, $b = -\max\{c(e) \mid e \in E\}$ and $B = 1 + \sum_{(\nu,i)\in N} w(\nu,i)$ are appropriate.

3.2.2. Facets

We will give now some intuition on the structure of the polyhedra for the implicit approach.

The polytope $P_{\mathrm{CGR}}^{\mathrm{impl,R}}$ is a full-dimensional polytope in $|N|(|E|+1)$-space. Again, $P_{\mathrm{CGR}}^{\mathrm{impl,S}}$ is obtained by cutting off integral corners from $P_{\mathrm{CGR}}^{\mathrm{impl,R}}$ with the capacity

constraints, and $P_{\mathrm{CGR}}^{\mathrm{impl}}$ is the convex hull of the integer solutions of $P_{\mathrm{CGR}}^{\mathrm{impl,S}}$. These polytopes are also full-dimensional.

Analogous observations can be made about the polyhedra for unconstrained global routing. $P_{\mathrm{UGR}}^{\mathrm{impl,R}}$ is a full-dimensional polyhedron in $|N||E|+1$-space, that is obtained by lifting the cross product of several Steiner tree polyhedra, one for each net, along the dimension x_L. $P_{\mathrm{UGR}}^{\mathrm{impl,S}}$ is obtained from $P_{\mathrm{UGR}}^{\mathrm{impl,R}}$ by cutting off parts with the load constraints. $P_{\mathrm{UGR}}^{\mathrm{impl}}$ is the convex hull of all integral points in $P_{\mathrm{UGR}}^{\mathrm{impl,S}}$. $P_{\mathrm{UGR}}^{\mathrm{impl,S}}$ and $P_{\mathrm{UGR}}^{\mathrm{impl}}$ are both full-dimensional polyhedra.

All facets of the polytopes $P_{\mathrm{CGR}}^{\mathrm{impl,R}}$ and $P_{\mathrm{UGR}}^{\mathrm{impl,R}}$ can be found by *lifting* facets from polytopes for some implicit formulation of the minimum Steiner tree problem, which is based on Steiner cuts. There are several such formulations in the literature together with discussions of the facets of the resulting polyhedra.[18,19,20]

As in the explicit approach, additional facets determine the structure of the sliced polyhedra and the actual routing polyhedra, in the implicit approach. The same problems exist for finding these facets, that were discussed in Sec. 3.1.2.

We omit the details of this construction, but note that both the full-dimensionality of the polytopes as well as the lifting theorem rely on the fact that the routing graph is biconnected.

There are also implicit formulations of the minimum Steiner tree problem that are not based on Steiner cuts.[21] These formulations can be used as a basis for global routing, as well.

4. Solution Methods

In this section we discuss several elements of finding good upper and lower bounds for global routing problems. We will emphasize methods for finding lower bounds, which have largely been lacking in this field. Many of the methods we propose have not been discussed before, in the context of global routing. Most of the methods result from applying well-known integer program techniques to global routing. We consider mainly the unconstrained global routing problem. Some of the solution methods can also be applied to constrained global routing.

4.1. Column Generation

If we use the explicit approach, a major problem is that there usually is an enormous number of admissible routes, each of which is represented by a separate variable. One way of dealing with this problem is, to limit oneself to few routes, e.g., in routing graphs that are grid graphs one can limit oneself to routes with few bends. Such a restriction reduces the size of the integer program and raises the hope of being able to optimize the program directly with integer programming methods. Raghavan and Thompson[10,11] have pursued this approach.

If we want to optimize over more admissible routes, we can use a method that is adapted from column generation techniques in linear programming. We start with a reasonable feasible routing $\overline{x}$. This routing can be obtained by a heuristic

or by optimizing the integer program on a restricted small set of admissible routes. Subsequently, we improve $\overline{x}$ by iteratively exchanging routes for single nets using the following procedure.

1. Let (ν, i) be an arbitrary net which, in routing $\overline{x}$, is routed by route $T^j_{\nu,i}$.
2. Let $E^*_{\nu,i}$ be the set of edges which are *improper bottlenecks* of that solution, e.g.,

$$E^*_{\nu,i} := \{e \in T^j_{\nu,i} \mid U(\overline{x}, e) - c(e) = \overline{x}_L\} \cup \{e \notin T^j_{\nu,i} \mid U(\overline{x}, e) - c(e) + w(\nu, i) \geq \overline{x}_L\}$$

3. Let $G_{\nu,i} := (V, E_{\nu,i})$ be the graph obtained by deleting all edges in $E^*_{\nu,i}$, e.g., $E_{\nu,i} := E \setminus E^*_{\nu,i}$.
4. Compute an arbitrary Steiner tree $T^{j'}_{\nu,i}$ on $G_{\nu,i}$.

Clearly, this exchange of route $T^j_{\nu,i}$ by route $T^{j'}_{\nu,i}$ decreases the cost of the routing. If, in Step 4, no route can be found for any net, we can consider *proper* bottlenecks in Step 2, e.g., edges from the set

$$E^*_{\nu,i} := \{e \notin T^j_\nu \mid U(\overline{x}, e) - c(e) + w(\nu, i) > \overline{x}_L\}$$

In this case, we can still hope to decrease the cost of the routing by finding near or exact minimum Steiner trees, but it can also be the case that all such exchanges increase the cost of the routing.

There are several ways of using this basic exchange step in optimization heuristics. On the one hand, it can be embedded into local search strategies for the optimum solution — either into a greedy strategy or in a global optimization procedure such as simulated annealing. On the other hand, we can fix the newly created route $T^{j'}_{\nu,i}$ and reiterate an optimization of the whole integer program with this side constraint.

4.2. Steiner Tree Algorithms

If we use the implicit approach, we have the additional problem, that we have to solve the NP-hard minimum Steiner tree problem. We will spend much effort on being able to route nets independently. Then routing reduces to computing independent Steiner trees, one for each net. Such methods were discussed in the past, under the notion of *sequential routing*. In classical sequential routing methods, however, the independence between nets is asserted heuristically, at the cost of losing the analyzability of the resulting routing with respect to the global optimum. In contrast, in Secs 4.4 and 4.5 we will aim at maintaining the analyzability of the quality of the resulting routings.

If the nets can be considered independent, then any of a large number of minimum Steiner tree algorithms can be applied, such as spanning tree enumeration algorithms,[22] topology enumeration algorithms,[22] dynamic programming

algorithms,[23] set covering algorithms,[24] implicit enumeration algorithms,[25] branch-and-bound algorithms, combined with Lagrange relaxation,[26,27] dual ascent algorithms,[28] and cutting-plane algorithms.[18,19,29] For a recent survey see Winter.[30]

4.3. Tradeoff Between the Explicit and the Implicit Approach

We have discussed the explicit and the implicit approach. In fact, these are only two ends of a whole spectrum of possibilities of describing routes. In this section we discuss another possibility of describing routes. Basically, we use enumeration. Instead of enumerating Steiner trees, however, we enumerate sets of nonrequired vertices. Let us denote such a set by $S^j_{\nu,i} \subseteq V \setminus \{(\nu,i)\}$ where j ranges from 1 to a number $J_{\nu,i}$. For each net (ν,i), we admit all routes which are spanning trees on one of the vertex sets $V \cup S^j_{\nu,i}$ for some j.

To formulate this approach as an integer program, we need to define two sets of variables. First we need, for every net $(\nu,i) \in N$ and for every set $S^j_{\nu,i}$, a variable $x_{\nu,i,j}$ which is set to 1, if net $(\nu,i) \in N$ is routed by a spanning tree on $V \cup S^j_{\nu,i}$, and which is set to 0 otherwise. Second we need, for each net $(\nu,i) \in N$, for each set $S^j_{\nu,i}$ and for each edge $e \in E$ a variable $x_{\nu,i,j,e}$ which is set to 1, if edge e is used to route net $(\nu,i) \in N$ with a spanning tree of $V \cup S^j_{\nu,i}$ and which is set to 0 otherwise.

We can base an integer program formulation of the global routing problem on the above definitions. This formulation explicitly enumerates all of the sets $S^j_{\nu,i}$, but implicitly describes all the spanning trees pertaining to set $S^j_{\nu,i}$ using some appropriate integer program formulation. By suitably defining the sets $S^j_{\nu,i}$, we may be able to trade off the problem of the enormous number of variables in the explicit approach with the problems that arise with the implicit approach to the minimum Steiner tree problem. For instance, we can introduce nontrivial admissible routes while still maintaining some degree of implicit route description.

4.4. Lagrange Relaxation

4.4.1. Introduction

Lagrange relaxation is a technique for computing lower bounds on the optimal cost of integer programs. It was applied with great success to the traveling salesperson problem[31,32] and to the minimum Steiner tree problem.[26,27] The basic idea is to split the constraints of the constraint matrix into two sets, one containing the constraints which are "easy" to meet and one containing the constraints which are "hard" to meet. In specific cases, the notion "easy" can mean, for instance, that the respective constraint matrix is totally unimodular or that the respective integer program can be solved efficiently, say, with graph-theoretic methods. For the traveling salesperson problem, arising "easy" problems are, for example, the 1-tree problem or the 2-matching problem, and for the minimum Steiner tree problem an easy problem that occurs is the minimum spanning tree problem.

In Lagrange relaxation, the hard constraints are eliminated, but violations of the hard constraints are penalized in the cost function. The penalty for each eliminated constraint is weighted with a constant factor, the so-called *Lagrange multiplier*. To be more precise, consider the following integer program:

$$
\begin{aligned}
(IP) \quad & \min \quad cx \\
& \text{s.t.} \quad Ax \leq a_0 && (1) \\
& \qquad Bx \leq b_0 && (2) \\
& \qquad x \geq 0 \\
& \qquad x \quad \text{integer} && (3)
\end{aligned}
$$

The constraint set (1) contains the "hard" constraints. The constraint set (2) contains the "easy" constraints. The following relaxed version of IP is called the *Lagrange problem.*

$$
\begin{aligned}
(LP_\lambda) \quad & \min \quad cx + \lambda(Ax - a_0) \\
& \text{s.t.} \quad Bx \leq b_0 \\
& \qquad x \geq 0 \\
& \qquad x \quad \text{integer}
\end{aligned}
$$

The vector $\lambda \geq 0$ contains the Lagrange multipliers, one for each constraint in set (1). Clearly, each feasible solution of IP is also a feasible solution of LP_λ, thus the optimum cost of LP_λ is a lower bound on the optimum cost of IP, for each assignment to the Lagrange multipliers. The goal is to find one such assignment that maximizes the optimum cost of LP_λ. The resulting maximization problem is called the *Lagrange dual LD.*

$$
\begin{aligned}
(LD) \quad & \max_{\lambda \geq 0} \min_{x} \quad cx + \lambda(Ax - a_0) \\
& \text{s.t.} \quad Bx \leq b_0 \\
& \qquad x \geq 0 \\
& \qquad x \quad \text{integer}
\end{aligned}
$$

The Lagrange dual can be solved with linear programming methods or with subgradient optimization. Subgradient optimization seems to be the more practical method. See Ref. 15 for more details on subgradient optimization.

The optimum cost of the Lagrange dual cannot be less than the optimum cost of the *linear relaxation* of IP, i.e., the problem obtained from IP by removing the constraints (3). This cost is attained by the Lagrange dual exactly if all corners of the polyhedron for the easy constraint set are integral.[15,16] Furthermore, Lagrange relaxation can be much more efficient due to the special shape of the Lagrange problems that have to be solved instead of applying general linear programming techniques.

4.4.2. Application to unconstrained global routing

For unconstrained global routing a natural set of constraints to eliminate are the load constraints, because these constraints are the only constraints that relate different nets to each other. If we consider the explicit approach, the resulting Lagrange problem has the following shape.

$$\begin{aligned} \min_{x} \quad & Z \cdot x_L + W(x) + \sum_{e \in E} \lambda_e (U(x,e) - c(e) - x_L) \\ \text{s.t.} \quad & \sum_{j=1}^{I_{\nu,i}} x_{\nu,i,j} = 1 \quad \text{for all nets } (\nu,i) \in N \\ & x_{\nu,i,j} \in \{0,1\} \quad \text{for all nets } (\nu,i) \in N \text{ and all } T^j_{\nu,i} \end{aligned} \tag{4}$$

In order for the relaxed program not to be unbounded, the Lagrange multipliers have to fulfill

$$\sum_{e \in E} \lambda_e = Z$$

Relaxing the load constraints has the effect that the Lagrange problem decomposes into a set of independent minimization problems, one for each net in N. A quick inspection of the Lagrange problem shows that the minimization problem for net (ν,i) is just a minimum Steiner tree problem, where the length of edge $e \in E$ is $\ell(e) + \lambda_e$.

If the explicit formulation is used, solving this problem just amounts to selecting the variable $x_{\nu,i,j}$ that represents the shortest route for net (ν,i). Since the constraint matrix of the Lagrange problem is totally unimodular, we cannot achieve better lower bounds with Lagrange relaxation than with linear relaxation. However, making the nets independent in Lagrange relaxation allows for an easy parallelization of the computation. This is one of the main advantages of this Lagrange relaxation over the computation of lower bounds by linear relaxation.

If the implicit formulation is used, the constraints (4) are replaced by the Steiner cut constraints. The result is an integer program that is hard to solve. However, we can apply Lagrange relaxation to this program, as well, to find good lower bounds,[26,27] or else we can try to apply algorithms that exactly solve the minimum Steiner tree problem. This is feasible, for instance, if all nets are two-terminal nets, because in this case this problem reduces to a shortest path problem.

The quality of Lagrange relaxations of the unconstrained global routing problems can be ascertained by proving statements on the size of the gap between the optimum cost of a routing and the optimum cost of its linear relaxation. Partial answers to this question exist. If only two-terminal nets are involved, $\ell \equiv 0$, and $Z = 1$, then the unconstrained global routing problem reduces to an integer multicommodity flow problem. In special cases of such problems it can be shown that the gap between the cost of the optimum routing, i.e., of the *integral multicommodity*

flow is at most 1 larger than the optimum cost of the linear relaxation, i.e., the cost of the *fractional multicommodity flow*.[33,34,35,36,37] If the gap is not too large, in general, then Lagrange relaxation is quite an accurate method of global routing.

4.5. Surrogate Relaxation

4.5.1. Introduction

Surrogate relaxation does not eliminate the set of hard constraints but merge them into a single constraint. Starting from the integer program *IP*, we form the following *surrogate program.*

$$\begin{aligned} (SP_\lambda) \qquad & \min \quad cx \\ & \text{s.t.} \quad \lambda Ax \leq \lambda a_0 \\ & \qquad Bx \leq b_0 \\ & \qquad x \geq 0 \\ & \qquad x \quad \text{integer} \end{aligned}$$

In the surrogate program, a nonnegative vector $\lambda \in \mathbb{R}^m$ is used to replace the $m > 1$ hard constraints by a single constraint. This technique is called *constraint aggregation.* Again, the optimal solution of the surrogate problem is a lower bound on the optimal solution of the original problem *IP*, no matter, what value of λ is chosen. We can maximize this bound by forming the *surrogate dual SD.*

$$\begin{aligned} (SD) \quad & \max_{\lambda \geq 0} \min_{x} \quad cx \\ & \text{s.t.} \quad \lambda Ax \leq \lambda a_0 \\ & \qquad Bx \leq b_0 \\ & \qquad x \geq 0 \\ & \qquad x \quad \text{integer} \end{aligned}$$

The lower bound provided by the surrogate dual can be better than the one obtained by Lagrange relaxation. In fact, for each $\lambda \geq 0$, we have[17]

$$c(LD_\lambda) \leq c(SD_\lambda)$$

Consequently, if L is the linear relaxation of *IP*, then we have

$$c(L) \leq c(LD) \leq c(SD) \leq c(IP)$$

4.5.2. Application to unconstrained global routing

In the case of unconstrained global routing, it is natural to merge the load constraints. Geometrically, this means that the load-roof of the unconstrained global routing polytope only consists of one hyperplane. Using different values in the aggregation vector λ, one can orient this hyperplane in a large number of ways.

The surrogate problem does not decompose into independent problems, one for each net, because the constraint obtained by merging the load constraints introduces a dependence between the nets. But, if we use the explicit approach, the

surrogate problem can be solved with linear programming methods or with greedy methods based on an exchange step such as discussed in Sec. 4.1: Each corner of its polyhedron is integral. The reason is that the polytope for the surrogate problem is obtained by lifting the full-dimensional integral polytope $P_{\mathrm{CGR}}^{\mathrm{expl,R}}$ (see Sec. 3.1.2) along the new dimension x_L and afterwards cutting it with a *single* hyperplane.

For global routing, surrogate relaxation cannot improve on the lower bounds obtained by Lagrange relaxation. But there is hope that, using surrogate relaxation, we can compute the bounds with more direct methods than subgradient procedures.

We are working on developing additional polyhedral methods for improving the lower bounds obtained by the relaxations described here. These methods are based on scanning the configuration space by successively forcing solutions to lie on specific faces of the relaxed polyhedra. For small examples, these methods are powerful enough to even obtain an optimum solution in very few steps. We report on these methods in a subsequent paper.

4.6. Timing Aspects

In some applications, additional design-rules besides the capacity or load constraints have to be observed. For instance, often certain nets have requirements on their minimum and maximum delay. If we consider the explicit approach, the integration of such timing criteria is quite simple. In fact, routes which violate the timing constraints are deemed inadmissible. Using the implicit approach, timing criteria are harder to ensure. But, for instance, in a model where each edge e on a route makes an additive contribution $t_{\nu,i,e}$ to the delay of net (ν, i), we can incorporate requirements on the minimum and maximum delay of a net.

For this purpose, let us assume that net (ν, i) has an upper bound of $d(\nu, i)$ on its delay. Then we can formulate the following integer program.

$$\min \beta x_T + Z \cdot x_L + W(x)$$

s.t.

$$\sum_{e \in E(X:V\setminus X)} x_{\nu,i,e} \geq 1 \text{for all } (\nu, i) \in N \text{ and all Steiner-cuts } (X : V \setminus X)$$

$$U(x, e) - c(e) \leq x_L \text{for all } e \in E$$

$$D(x, \nu, i) - d(\nu, i) \leq x_T \text{for all } (\nu, i) \in N$$

$$-\max\{c(e) \mid e \in E\} \leq x_L \leq \eta$$

$$-\max\{d(\nu, i) \mid (\nu, i) \in N\} \leq x_T \leq \vartheta$$

$$x_{\nu,i,e} \in \{0, 1\}$$

Here we use the following additional linear function.

Delay of net (ν, i)

$$D(x, \nu, i) = \sum_{e \in E} t_{\nu,i,e} x_{\nu,i,e} \text{for every } (\nu, i) \in N$$

The constants β and Z are chosen such as to trade off the contributions of the different terms in the cost function appropriately for the application. The constants η and ϑ are appropriate constants that bound the polyhedra involved.

Lower bounds on delays can be incorporated in a similar fashion.

5. Integrating Placement and Global Routing

The placement/floorplanning process precedes global routing. The goal of placement/floorplanning is to distribute the circuit components over the chip in such a way that wiring is possible in small space. The placement process is hampered severely by the fact that it is very difficult to come up with easy-to-compute and accurate estimates of wiring area. Thus, recently, research has been directed towards integrating placement with global routing, i.e., the router itself is used to provide wiring estimates. Heuristic versions of this approach have been presented in Refs. 9, 38, 39, 40.

In this section, we will integrate placement and global routing on the basis of integer programming. The result will be a framework, in which it is possible, in principle, to solve the placement and global routing problem *exactly* using integer programming methods.

5.1. The Integer Program

We will coach our problem in terms of a simple version of gate-array layout. In gate array-layout we have available a number of *slots* in which we can place *gates*, such as NOR-gates or flip-flops. The slots are arranged on the chip surface in some fashion. Each gate can be placed into one of a certain subset of the slots. We call these slots the *admissible* slots for the gate. For simplicity, we will assume that each slot can hold no more than one gate. The arrangement of slots on the chip surface gives rise to a specific routing graph $G = (V, E)$ whose vertices are the slots and whose edges are the connecting wiring channels. The routing graph is the abstraction of the *gate-array master* for our problem. Figure 4 shows a regular arrangement of slots across a chip surface in the so-called *island style*.

An instance of the *placement problem* is a circuit and a routing graph. The circuit is a network of gates, connected by wires. Formally, the circuit is a hypergraph $C = (B, N)$, whose vertices $b \in B$ are the gates and whose hyperedges $(\nu, i) \in N$ are the wires. The task is to assign the gates to slots in such a manner that the wiring complexity be minimized. The global routing problem then finds the optimum wiring, given a placement. We want to solve both problems simultaneously. So, the objective is to come up with the assignment of the gates to slots that minimizes the cost of the global routing *and* to compute the corresponding optimum routing. We can use either the constrained or the unconstrained version of the global routing problem to find and evaluate the routing.

Let us coach this task in terms of an integer program. We do so by extending the global routing integer programs discussed in Sec. 2. It is largely immaterial which

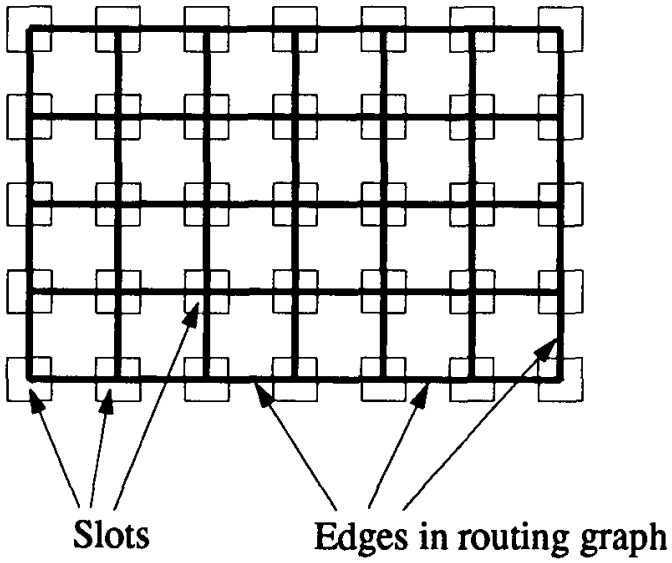


Fig. 4. A rectangular arrangement of slots and the resulting routing graph.

of the integer programs in that section we choose. For purposes of illustration, let us choose $I^{\text{expl}}_{\text{UGR}}$.

For simplicity of the exposition, we assume that the number of slots equals the number gates, and that all slots are admissible for all gates.

We add new variables to the integer program. The binary variable x_{bv} is supposed to be 1, if the gate b is placed on slot v, and 0 otherwise. The following *placement constraints* ensure that a bijection is formed from the set of gates to the set of slots.

$$\sum_{b \in B} x_{bv} = 1 \quad \text{for all } v \in V$$

$$\sum_{v \in V} x_{bv} = 1 \quad \text{for all } b \in B$$

In addition we need a set of constraints that ensures that the routing is consistent with the placement, i.e., that the route for net (ν, i) terminates at the slots holding the terminals of net (ν, i). We will do so in a somewhat redundant fashion for reasons that will become clear later.

By a *fixation* for net (ν, i) we mean an injective mapping $f_{\nu,i}$ that maps the terminals of net (ν, i) to slots. For each fixation $f_{\nu,i}$ we introduce a variable $x_{f_{\nu,i}}$ which is supposed to be 1 if the fixation $f_{\nu,i}$ is realized by the placement, and 0 otherwise. This interpretation of the variable $x_{f_{\nu,i}}$ is ensured by the following *primary nailing constraints.*

$$\left(\sum_{\substack{f_{\nu,i} \text{ fixation for } (\nu,i) \\ \text{that maps gate} \\ b \text{ onto slot } v}} x_{f_{\nu,i}} \right) - x_{bv} = 0 \quad \text{for all } (\nu,i) \in N,\ b \in (\nu,i),\ v \in V$$

Furthermore, exactly one fixation should be chosen for each net, resulting in the *fixation completeness constraints*

$$\sum_{f_{\nu,i} \text{ fixation for } (\nu,i)} x_{f_{\nu,i}} = 1 \quad \text{for all } (\nu,i) \in N$$

Not all fixations need be admissible. In the integer program, we only provide variables for the admissible fixations.

By a *route* for net (ν,i) we mean an admissible Steiner tree that has $|(\nu,i)|$ terminals. These terminals can be assigned to slots in an arbitrary way. We again enumerate the admissible routes for net (ν,i) with $T^1_{\nu,i}, \ldots, T^{I_{\nu,i}}_{\nu,i}$. Each route receives a variable $x_{\nu,i,j}$, as in Sec. 2. (Note, however, that this time there are even more admissible routes for each net than in Sec. 1.) Net (ν,i) has to be wired with exactly one route, giving rise to the *routing completeness constraints* which take exactly the same form as the completeness constraints in Sec. 2.

$$\sum_{j=1}^{I_{\nu,i}} x_{\nu,i,j} = 1 \quad \text{for all } (\nu,i) \in N$$

In addition, we have to ensure, that the route chosen for net (ν,i) is consistent with the fixation chosen for net (ν,i). This is the purpose of the *secondary nailing constraints*

$$\left(\sum_{\substack{T^j_{\nu,i} \text{ is a route} \\ \text{that realizes } f_{\nu,i}}} x_{\nu,i,j} \right) - x_{f_{\nu,i}} = 0 \quad \text{for all } (\nu,i) \in N,\ f_{\nu,i} \text{ fixation for } (\nu,i)$$

Finally, we need the load constraints, as in Sec. 2. The cost function is identical to that in Sec. 2.

The fixation is a notion that is not really necessary for an integer program formulation of our problem. But introducing fixations enables us to distinguish between different routes for the same net, that differ only in the permutation of the terminals. Such routes should be distinguished in optimization algorithms because they imply different placements. In concise integer programs, such routes would be represented by the same variable and thus could not be distinguished, however.

The resulting IP is denoted with I_{place}. It is exceedingly large. It has both an exponential number of variables and constraints. Nevertheless, we will show in the next section, that it can be handled.

5.2. A Lagrange Relaxation

We will relax the following constraint sets:

- The primary nailing constraints
- The load constraints

The second group of constraints is already known to us as a good candidate for a relaxation, from Sec. 4. The first group allows for splitting off the placement constraints (involving the x_{bv}), from the rest of the integer program (involving the the $x_{f_{\nu,i}}$ and the $x_{\nu,i,j}$).

Let us discuss what the cost function of the Lagrange problem looks like. Each of the primary nailing constraints is associated with a Lagrange multiplier $\lambda_{b,v,\nu,i} \in \mathbb{R}$. Each of the load constraints is associated with a Lagrange multiplier $\lambda_e \geq 0$. In addition to the original terms in c_{UGR} the cost function of the Lagrange problem contains the following terms which arise from the relaxation.

$$\left(- \sum_{\substack{(\nu,i)\in N \\ b\in(\nu,i)}} \lambda_{b,v,\nu,i} \right) x_{bv} \quad \text{for all } v \in V, b \in B \tag{5}$$

$$\left(\sum_{\substack{f_{\nu,i} \text{ maps} \\ b \text{ onto } v}} \lambda_{b,v,\nu,i} \right) x_{f_{\nu,i}} \quad \text{for all nets } (\nu,i) \text{ and fixations } f_{\nu,i} \tag{6}$$

$$\left(w(\nu,i) \sum_{e\in T^j_{\nu,i}} \lambda_e \right) x_{\nu,i,j} \quad \text{for all nets } (\nu,i) \text{ and admissible routes } T^j_{\nu,i} \tag{7}$$

We have to choose the λ_e such that $\sum_{e\in E} \lambda_e = Z$, otherwise the relaxed problem is unbounded.

The Lagrange problem decomposes into two independent parts.

First, the placement constraints give rise to a classical linear assignment problem with a cost function that is determined by the factors in (5).

Second, the part of the Lagrange problem encompassing the variables $x_{\nu,i,j}$ and $x_{f_{\nu,i}}$, and the secondary nailing constraints as well as the routing completeness constraints is a set of independent nonstandard Steiner tree problems, one for each net (ν,i). Given (ν,i), the problem is to find an admissible Steiner tree with $|(\nu,i)|$ (arbitrarily placed) terminals that minimizes a cost which is composed additively of two terms. The first term is the sum of the lengths of all edges e in the tree.

Here, the length of edge e is additively composed from the edge length in G and from the factor for e in (7). The second term is the sum of the costs of the terminal assignments to slots. Here each assignment of a gate b to a slot v is weighed with the corresponding contribution $\lambda_{b,v,\nu,i}$ from (6). This factor may be negative.

This problem can be transformed into a standard minimum Steiner tree problem as follows (see Fig. 5). Adjoin new vertices $b_1, \ldots, b_{|(\nu,i)|}$ to the routing graph G and connect them to V with a complete bipartite graph, such that the edge $\{b_j, v\}$ is labeled with $\lambda_{b_j,v,\nu,i}$. Add the same suitable constant Γ to all edges in the complete bipartite graph, such that each of these edges receives labels that are larger than the total edge weight in G. Then solve the minimum Steiner tree problem with the $b_1, \ldots, b_{|(\nu,i)|}$ as terminals. (Adding Γ to the edges in the complete bipartite graph ensures that exactly $|(\nu, i)|$ such edges are included in any minimum Steiner tree.)

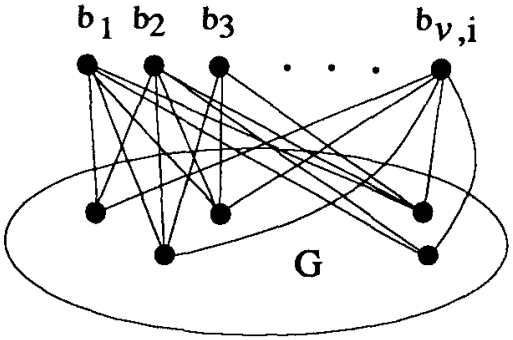


Fig. 5. The reduction of the nonstandard Steiner tree problem to a minimum Steiner tree problem.

5.3. Discussion and Example

Let us, for convenience, assume that all edge lengths in G are zero, i.e., only the maximum load is considered in the cost function of the original problem and $Z = 1$. Setting all $\lambda_{b,v,\nu,i} = 0$, and $\lambda_e = 1/|E|$ leads to a Lagrange problem whose optimum cost is the *minimum average edge load*: Select, for each net the route which induces the smallest overall load in G, add up all loads and divide by the number of edges in G. The resulting cost is the optimum cost of the Lagrange problem. Clearly, this figure is a lower bound on the optimum cost of the original problem. If nets are evenly distributed in the circuit and the routing graph is homogeneous, then this lower bound will, in fact, be quite tight. However, we can do better. In fact, this assignment to the Lagrange multipliers, by setting all $\lambda_{b,v,\nu,i}$ to zero, disregards the dependency between different nets, via the incidences in the circuit C. The following example shows that, in general, setting the $\lambda_{b,v,\nu,i}$ to values that are different from zero, improves upon this lower bound.

Consider the problem instance in Fig. 6. The edges have capacities 1 and M, respectively, wehere M is an appropriately large number.

The admissible fixations for net ν_1 are depicted in Fig. 7. They are exactly those fixations that place the two terminals of the net on adjacent vertices in G. The fixations for net ν_2 are defined analogously.

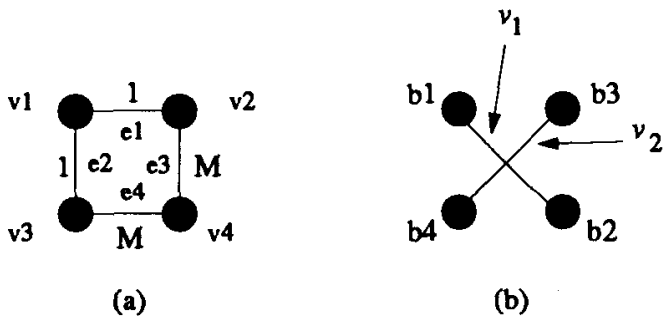


Fig. 6. A problem instance. (a) The routing graph. Edges are labeled with their capacities. (b) The circuit.

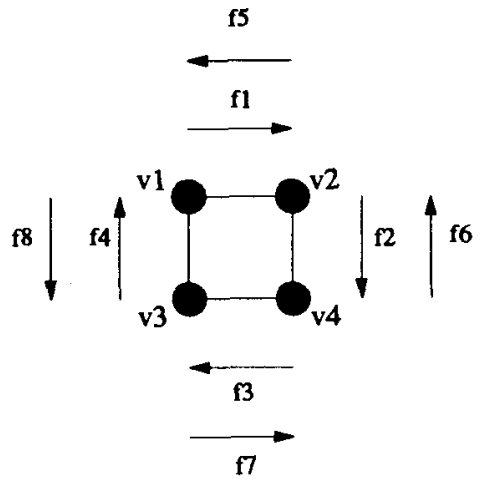


Fig. 7. The eight fixations for net ν_1. Each fixation is represented by an arrow between two vertices of G. The tail of each arrow corresponds to gate b_1 and the head to gate b_2.

An optimal solution places gate i on vertex i, for $i = 1, \ldots, 4$. Its cost is 0.

Choosing the above assignment to the Lagrange multipliers yields an optimum cost of $-M/2$. In all optimal solutions with respect to this relaxation each of the two nets is routed along a high-capacity edge. Each such solution violates the incidences in the circuit C, however. The best possible assignment to the λ_{e_i}, if the $\lambda_{b,\nu,\nu,i}$ are all set to zero, is $\lambda_{e_1} = \lambda_{e_2} = 1/2$ and $\lambda_{e_3} = \lambda_{e_4} = 0$. The resulting cost of an optimal solution is -1.

In order to account for incidences in C, we set $\lambda_{b,1,\nu,i} = -1/2$, for all possible choices of b and (ν, i), i.e., we give a weight of $-1/2$ to assignments of all gates to vertex 1. Furthermore, we set $\lambda_{e_1} = \lambda_{e_2} = 1/2$ and $\lambda_{e_3} = \lambda_{e_4} = 0$, as before. The

respective optimal cost is $-1/2$: The cost contribution of an optimal routing is -1. To this value we have to add $1/2$ for the cost of an optimal linear assignment of gates to vertices.

The optimum cost of the linear relaxation for this example is also $-1/2$.

6. Summary

Global routing is an interesting application of integer programming. Generic global routing problems can be located somewhere in the middle between the highly abstracted problems that have traditionally been popular as test cases for combinatorial optimization, such as the traveling salesperson problem, and the more unstructured problems that occur in practice. Therefore, research on global routing can be a valuable contribution to the transfer of advanced optimization methods into practical settings. Many of the methods suggested in this article still have to prove themselves in practice. We are currently in the process of implementing several of these methods on sequential as well as parallel computers and streamlining them using experiments on real-world data.

Acknowledgement

Part of this research was conducted while the authors were attending a research initiative at the Fibonacci Institute, Trento, Italy in summer 1991.

References

1. T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout*, Teubner–Wiley Series of Applicable Theory in Computer Science (John Wiley & Sons, New York, 1990).
2. R. Nair, Personal communication, 1991.
3. M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness* (W. H. Freeman and Company, San Francisco, CA, 1979).
4. M. R. Kramer and Jan van Leeuwen, "The complexity of wire routing and finding minimum area layouts for arbitrary VLSI circuits", ed. F. P. Preparata, in *Advances in Computing Research, Volume 2: VLSI Theory*, JAI Press, Reading, MA, 1984, pp. 129–146.
5. B. Korte, H. J. Prömel and A. Steger, "Steiner trees in VLSI-layout", in *Paths, Flows and VLSI-Layout, Algorithms and Combinatorics Volume 9* (Springer Verlag, Heidelberg, Germany, 1990) pp. 185–214.
6. M. Burstein and R. Pelavin, "Hierarchical wire routing", *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems* **CAD-2(4)** (1983) 223–234.
7. W. K. Luk, P. Sipala, M. Tamminen, D. Tang, L. S. Woo and C. K. Wong, "A hierarchical global wiring algorithm for custom chip design", *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems* **CAD-6(4)** (1987) 518–533.
8. M. Burstein and S. J. Hong, "Hierarchical VLSI layout: Simultaneous wiring and placement", eds. F. Anceau and E. J. Aas, in *Proc. of VLSI'83* (Elsevier Science Publishers B. V., Amsterdam, The Netherlands, 1983) pp. 45–60.
9. T. Lengauer and R. Müller, "A robust framework for hierarchical floorplanning with integrated global wiring", in *Proc. of the Int. Conf. on Computer-Aided Design* (IEEE, 1990) pp. 148–151.

10. P. Raghavan and C. D. Thompson, "Randomized rounding: A technique for provably good algorithms and algorithmic proofs", *Combinatorica* **7(4)** (1987) 365–374.
11. A. P.-C. Ng, P. Raghavan and C. D. Thompson, "Experimental results for a linear program global router", *Computers and Artificial Intelligence* **6(3)** (1987) 229–242.
12. P. Raghavan, "Probabilistic construction of deterministic algorithms: Approximating packing integer programs", *J. Comput. and Syst. Sci.* **37(2)** (1988) 130–143.
13. R. Nair, C. L. Berman, P. S. Hauge and E. J. Yoffa, "Generation of performance constraints for layout", *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems* **CAD-8(8)** (1989) 860–874.
14. E. L. Lawler, J. K. Lenstra, A. H. G. R. Kan and D. B. Shmoys, *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*, Wiley-Interscience Series in Discrete Mathematics (John Wiley & Sons, New York, 1985).
15. G. L. Nemhauser and L. A. Wolsey, *Integer and Combinatorial Optimization*, (John Wiley & Sons, New York, 1988).
16. A. Schrijver, *Theory of Linear and Integer Programming*, Wiley-Interscience Series in Discrete Mathematics and Optimization, (John Wiley & Sons, New York, NY, 1986).
17. S. Walukiewicz, *Integer Programming, Mathematics and Its Applications Volume 46* (Kluwer Academic Publishers, Bostin, MA, 1991).
18. S. Chopra and M. R. Rao, "The Steiner tree problem I: Formulations, compositions and extension of facets", Technical report, Graduate School of Business Administration, New York University, New York, NY, 1988.
19. S. Chopra and M. R. Rao, "The Steiner tree problem II: Properties and classes of facets. Technical report, Graduate School of Business Administration, New York University, New York, NY, 1989.
20. M. Grötschel and C. L. Monma, "Integer polyhedra arising from certain network design problems with connectivity constraints", Technical Report 104, Institute of Mathematics, University of Augsburg, Augsburg, Germany, 1988.
21. W. G. Liu, "A lower bound for the Steiner tree problem in directed graphs", *Networks* **20** (1990) 765–778.
22. S. L. Hakimi, "Steiner's problem in graphs and its implications", *Networks* **1** (1971) 113–133.
23. S. E. Dreyfus and R. A. Wagner, "The Steiner problem in graphs", *Networks* **1** (1972) 195–208.
24. Y. P. Aneja, "An integer linear programming approach to the Steiner problem in graphs", *Networks* **10** (1980) 167–178.
25. M. L. Shore, L. R. Foulds and P. B. Gibbons, "An algorithm for the Steiner problem in graphs", *Networks* **12** (1982) 323–333.
26. J. E. Beasley, "An algorithm for the Steiner problem in graphs", *Networks* **14** (1984) 147–159.
27. J. E. Beasley, "An SST-based algorithm for the steiner problem in graphs", *Networks* **19** (1989) 1–16.
28. R. T. Wong, "A dual ascent approach for Steiner tree problems on a directed graph", *Mathematical Programming* **28** (1984) 271–287.
29. S. Chopra, E. Gorres and M. R. Rao, "Solving the Steiner tree problem on a graph using Branch and Cut", *ORSA Journal on Computing*, 1991. To appear.
30. P. Winter, "Steiner problem in networks: A survey", *Networks* **17** (1987) 129–167.
31. M. Held and R. M. Karp, "The traveling-salesman problem and minimum spanning trees", *Operations Research* **18(6)** (1970) 1138–1162.
32. M. Held and R. M. Karp, "The traveling-salesman problem and minimum spanning trees: Part II", *Mathematical Programming* **1** (1971) 6–25.

33. A. V. Karzanov, "Half-integral five-terminaus flows", *Discrete Applied Mathematics* **18** (1987) 263–278.
34. A. V. Karzanov, "Path and metrics in a planar graph with three or more holes I", Technical Report RR 816-M-, IMAG ARTEMIS, Grenoble, France, 1990.
35. A. V. Karzanov, "Path and metrics in a planar graph with three or more holes II", Technical Report RR 817-M-, IMAG ARTEMIS, Grenoble, France, 1990.
36. H. Okamura, "Multicommodity flows in graphs", *Discrete Applied Mathematics* **6** (1983) 55–62.
37. H. Okamura and P. D. Seymour, "Multicommodity flows in planar graphs", *J. of Combinatorial Theory, Series B* **31(1)** (1981) 75–81.
38. W. W.-M. Dai and E. S. Kuh, "Simultaneous floorplanning and global routing for hierarchical building block layout", *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems* **CAD-6(5)** (1987) 828–837.
39. G. Zimmermann, "Top-down design of digital systems", ed. E. Hörbst, in *Advances in CAD for VLSI, Volume 2: Logic Design and Simulation* (North-Holland, New York, 1986) pp. 185–206.
40. G. Zimmermann, "A new area and shape function estimation technique for VLSI layouts", in *Proc. of the 25th Design Automation Conf.* ACM/IEEE, 1988, pp. 60–65.

Algorithmic Aspects of VLSI Layout
Eds. M. Sarrafzadeh and D. T. Lee
© World Scientific Publishing Co.

CIRCUIT PARTITIONING ALGORITHMS BASED ON GEOMETRY MODEL*

TETSUO ASANO
Department of Applied Electronics, Osaka Electro-Communication University
18-8 Hatsu-cho, Neyagawa, 572 JAPAN
e-mail: asano@efreet.osakac.ac.jp

and

TAKESHI TOKUYAMA
IBM Research, Tokyo Research Laboratory, 1623-14 Shimotsuruma,
Yamato, 242, JAPAN
e-mail: ttoku@vnet.ibm.com

ABSTRACT

This paper is concerned with the problem of partitioning a set of modules in VLSI circuit into two parts so that the number of interconnections between different sides is minimized while the total areas of modules are comparable in the two sides. Since the problem itself seems to be intractable, we present approximation algorithms. Given a set of nets and modules to be laid, we first calculate connectivity for each pair of modules to build a graph weighted by connectivities between modules. Then, we map those vertices onto points in the plane so that squared distances between points are roughly anti-proportional to the weights (connectivities) between vertices corresponding to the points. Finally, we try to find a linear partition by a straight line that achieves the minimum number of interconnections between two different sides determined by the partition.

Keywords: Circuit partitioning problem, duality transform, topological walk algorithm, arrangement of lines, Computational Geometry.

1. Introduction

A number of methods have been proposed for determining placement of modules in VLSI chip. One of the most popular methods is "Mincut Placement Algorithm" proposed by Lauther in 1980 [12]. The basic strategy of this algorithm is so-called divide-and-conquer. That is, a set of modules is divided into two parts so that the number of interconnections between different sides is minimized under the constraint that the total areas are comparable in the two sides. Then, an optimal

*This work was partially supported by Grant in Aid for Scientific Research of the Ministry of Education, Science and Cultures of Japan.

placement of modules is determined in each side in a recursive manner. Finally, the interconnections between the two sides are completed.

This Mincut Algorithm seems to be very promising if we can find an optimal partition of modules. Unfortunately the problem seems to be intractable [10]. This is mainly because of exponentially many different partitions. Two different approaches may be considered. In one approach we rely on heuristic search toward an approximated goal. In this case, for example, maximization of the minimum connectivity among modules in the same part is one such approximated goal. The other approach is to find a partition that is best in a restricted search space.

The above described problem of partitioning a set of modules into two parts so as to minimize the number of nets between different sides has usually been formulated using a graph representation. In one such representation vertices correspond to modules and an edge between two vertices has a weight proportional to the connectivity between the two corresponding modules. Then, an approximated goal may be to find a partition that maximizes the minimum connectivity between modules in the same side.

An alternative approach we will present is totally different. We first compute connectivities among modules as before. Then, we map modules into points in the plane in such a way that the distance between any two points is anti-proportional to the connectivity between the corresponding modules as much as possible. Thus, two tightly connected modules should be placed close to each other. Furthermore, we put a restriction on partitions, that is, we only consider partitions by straight lines (called linear partitions). A combinatorial observation tells us the fact that there are only $O(n^2)$ different linear partitions, which allows us to examine all possible linear partitions in polynomial time. One more nice thing about it is that such exhaustive search can be carried out in an efficient way using Geometric Transform [4, 14] that maps points into lines and lines into points and Topological Walk Algorithm [3] for searching in the transformed plane.

An efficient algorithm for finding an optimal linear partition is presented. It runs in $O(n^2 + M)$ time where n is the number of modules and M is the length of a net list. In many practical cases we may have constraints that some modules (pads) must be placed in predetermined sides. In this case the number of feasible linear partitions may happen to be much smaller than $O(n^2)$. When there are K feasible partitions the algorithm runs in $O(n \log n + K + M)$ time. Moreover, it can deal with so-called critical nets by assigning them large weights.

We could find a partition of a point set to minimize the larger diameter of the resulting sets [2]. In this case it is known that there is an optimal partition which is linear. This makes it somewhat reasonable for us to consider linear partitions alone.

2. Problem Formulation

In this section we prepare several terminologies and notations.

Let S be a set of modules $m_1, m_2, \ldots, m_n$. A net N_j is specified as a set of modules to be interconnected. Although it is usually given as a set of terminals there is no need here to know which terminals to be interconnected. We assume that there is no net interconnecting terminals of only one module. Area of a module m_i is denoted by area(m_i). For a set A of modules the sum of the areas of the constituent modules is denoted by area(A), that is,

$$\text{area}(A) = \sum_{m_i \in A} \text{area}(m_i) .$$

Now we are ready to define our problem.

Circuit Partitioning Problem

[INSTANCE]

(1) A set of modules $S = \{m_1, m_2, \ldots, m_n\}$.
(2) Area area(m_i) of each module $m_i, i = 1, 2, \ldots, n$.
(3) A set of nets (net list).
(4) A real number $r, 0 \leq r < 1$, which is a limit on the relative difference of areas of two parts.

[QUESTION]

Is there a partition (S_1, S_2) of S such that the number of nets interconnecting modules of different sides, that is, $|\{N_j | N_j \cap S_1 \neq \phi \text{ and } N_j \cap S_2 \neq \phi\}|$, is minimized under the constraint

$$\frac{|\text{area}(S_1) - \text{area}(S_2)|}{\text{area}(S)} \leq r ,$$

or equivalently

$$\frac{1-r}{2}\text{area}(S) \leq \text{area}(S_1) \leq \frac{1+r}{2}\text{area}(S) .$$

The above condition is referred to as **area constraint**.

Unfortunately, the problem is known to be NP-complete (see [10]), which means that there seems to be no efficient algorithm for finding an optimal solution for this problem. Since we need an efficient (polynomial-time) algorithm, we must either try to achieve an approximated goal by some heuristic algorithm or find an optimal solution in a restricted search space. Several different approximated goals may be considered. Many of them are concerned about the notion of connectivity or dissimilarity between modules. For two modules their connectivity is high if there are

many nets interconnecting them. The dissimilarity is anti-proportional to connectivity. Then, approximated goals may be to maximize the minimum connectivity in each side, maximizing the sum of connectivities between modules in the same side, minimizing the maximum dissimilarity in each side, and so on.

Example. An example is shown in Fig. 1, which consists of 17 modules/pads denoted by a through q and 24 nets. Area of each module is shown below.

module	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q
area	2	2	2	2	2	2	6	20	6	9	16	12	9	6	16	8	25

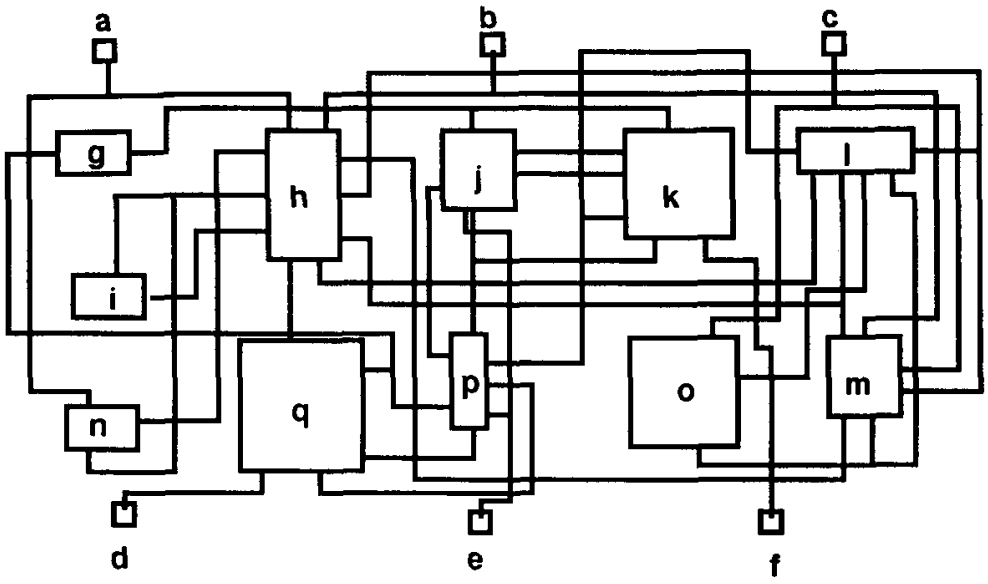


Fig. 1. An example with 17 modules/pads and 24 nets.

The net list is given as a set of nets expresses as sets of modules to be interconnected. Positions of terminals are not interested here. An optimal bipartition of the module set when the relative difference of the total areas in two sides is specified as 10% is shown in Fig. 2. In the partition the sum of areas of modules in the upper part is 75 while that for the lower part is 70.

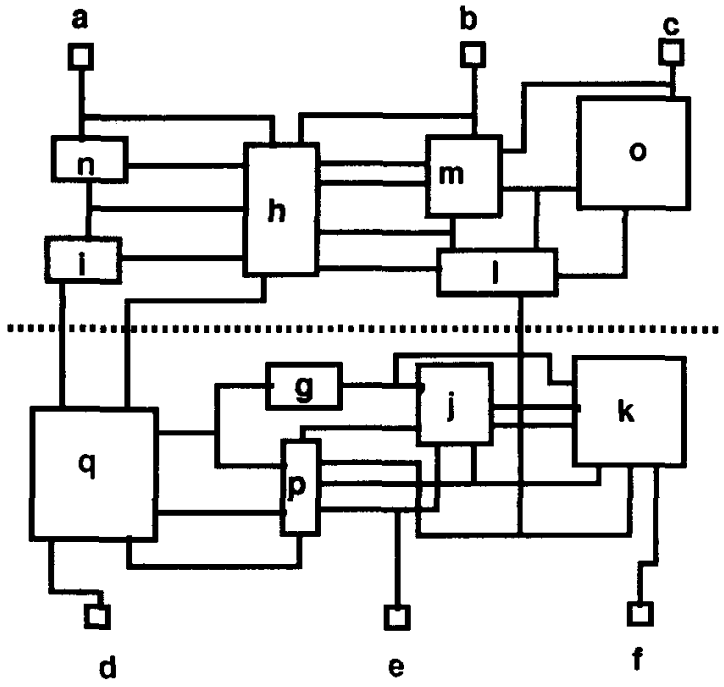


Fig. 2. An optimal bipartition when $r < 0.1$.

3. Algorithms Based on Geometry Model

3.1. Geometry Model

One natural way of formulation of the problem may be to calculate connectivities among modules to build a "connectivity" graph. In some case transitive property may be taken into accounts. In the geometry model to be described in this paper we do not build a graph but map modules into points in such a way that two tightly interconnected modules (those with many common nets) be laid close to each other. There are several ways for such geometric embedding (Refer to Hall [11], Donath and Hoffman [6], Tsay and Kuh [16], Frankle and Karp [8], Donath [5], Sigl, Doll, and Johannes [15], etc.) In this paper we are not interested in which of those methods is most effective for our purpose but in how efficiently we can find a best bipartition of the point set. So, we simply assume that modules are mapped into points in the plane according to some method and tightly connected modules are laid close to each other.

3.2. Problem Definitions

Suppose we have mapped the modules into points in the plane. Then, an approximated goal is to find a bipartition of the point set so as to minimize the larger diameter of the resulting sets. An algorithm for finding such an optimal bipartition of a point set is presented by Asano, *et al.* [2]. The algorithm runs in $O(n \log n)$ time.

One important observation here is that in the problem of finding a partition to minimize the larger diameter among optimal bipartitions there is one determined by a straight line (such a partition is called a linear partition). This observation suggests us that restriction to linear partitions may not be so harmful to a more general goal, that is, even if we restrict partitions only to linear ones we can expect in many cases that an optimal partition to minimize the number of interconnections between two sides may be obtained.

The following natural question comes up.

Problem G1. Given a set of points corresponding to modules in the plane, find a **linear partition** of S into two disjoint subsets that minimizes the number of nets interconnecting modules corresponding to points separated by the partition under some area constraint determined by a parameter $r, 0 < r < 1$.

Moreover, in practical cases some modules (or pads) should be placed in predefined sides. Then, the problem becomes still more complicated.

Problem G2. Solve the Problem G1 when some modules must be placed in predetermined sides.

Problem G3. Solve the Problem G2 in the presence of some critical nets.

We shall show that all of the above problems can be solved efficiently using Topological Walk Algorithm on arrangement of lines devised by the authors [3].

3.3. Problem G1: Optimal Linear Partition

The key idea behind our algorithm to be presented is the duality transform between points and lines. We map a point $p = (a, b)$ into the line T_p: $y = ax + b$ in the dual plane and a line L: $y = kx + d$ into a point T_L: $(-k, d)$. One important property of the duality transform is that it keeps the vertical relationship between points and lines. That is, if a point p lies above (below, respectively) a line L then the corresponding line T_p passes above (below, resp.) the corresponding point T_L. We first map a set of points in the plane into a set of lines in the dual plane. Then, those lines partition the plane into many small regions (called cells).

Figure 3 shows a simple example consisting of four modules and four nets together with an arrangement of four corresponding lines. Here note that each cell can be characterized by vertical position with respect to each line. In the figure, the cell, for example, which lies below the lines 1, 2, and 3 and above the line 4 is symbolically characterized as $B(1) \cap B(2) \cap B(3) \cap A(4)$. By the property of the duality transform an arbitrary point in this cell corresponds to a line in the primal plane such that it passes below the points 1, 2, 3 and above the point 4, which means that the point set $\{1, 2, 3, 4\}$ is separated into $\{1, 2, 3\}$ and $\{4\}$ by the line. Therefore, if we examine all the cells in the dual plane, it means we have examined all possible linear partitions of the original point set.

In this section we present an algorithm for finding an optimal linear partition such that the number of nets interconnecting different sides is minimum while it

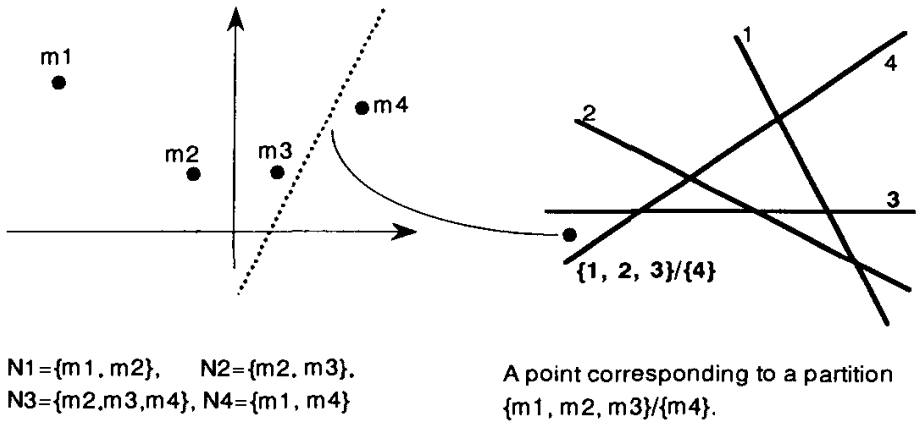


Fig. 3. Duality transform.

satisfies a given area constraint. First of all we make the following two assumptions which will be removed later:

(1) Every net is a (so-called) pin-pair which interconnects exactly two modules.
(2) Any two nets are different as sets.

In the following modules and corresponding points are identified. So, the point corresponding to a module m_i is also denoted by m_i.

Suppose we partition the set S of points by a line of angle θ. As is shown in Fig. 4, projection of those points onto a line of angle $\theta + \pi/2$ determines the order of the points. Let $(m_{\sigma(1)}, m_{\sigma(2)}, \ldots, m_{\sigma(n)})$ be such an order. Then, for every $i, 1 \leq i \leq n-1$, we have partition $\{m_{\sigma(1)}, \ldots, m_{\sigma(i)}\}/\{m_{\sigma(i+1)}, \ldots, m_{\sigma(n)}\}$. In addition, there are two trivial partitions ϕ/S and S/ϕ, both of which can be excluded since they do not satisfy any area constraint.

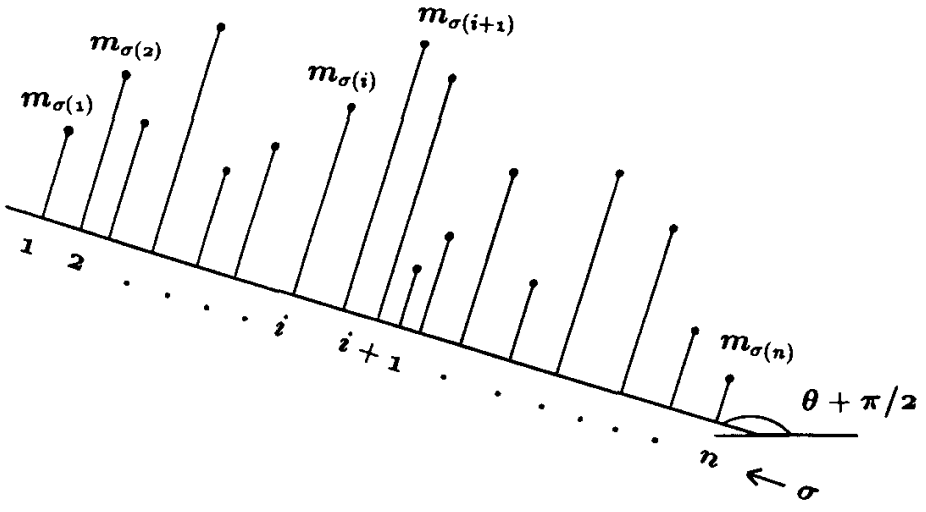


Fig. 4. Projetion of points onto a line of angle θ and its associated ordering σ.

In order to evaluate a partition $\{m_{\sigma(1)}, \ldots, m_{\sigma(i)}\}/\{m_{\sigma(i+1)}, \ldots, m_{\sigma(n)}\}$ we define the following notations.

$$\min(N_i, \sigma) = \min\{j | m_{\sigma(j)} \in N_i\} ,$$
$$\max(N_i, \sigma) = \max\{j | m_{\sigma(j)} \in N_i\} .$$

That is, if a net N_i interconnects points p and q of ranks j and k in the order σ, $j < k$, then $\min(N_i, \sigma)$ represents the smaller rank j and $\max(N_i, \sigma)$ the greater rank k.

Then, fixed an order σ, for each point $m_{\sigma(i)}$ we can compute the number of nets going up and going down from the point, denoted by $d_a(i, \sigma)$ and $d_b(i, \sigma)$, respectively. Formally they are defined by

$$d_a(i, \sigma) = |\{N_j | \max(N_j, \sigma) = i\}| ,$$
$$d_b(i, \sigma) = |\{N_j | \min(N_j, \sigma) = i\}| .$$

Based on the above information, the number of nets interconnecting different sides for the partition $\{m_{\sigma(1)}, \ldots, m_{\sigma(i)}\}/\{m_{\sigma(i+1)}, \ldots, m_{\sigma(n)}\}$, denoted by $\lambda_\sigma(i, i+1)$, is determined as follows (see Fig. 5).

$$\lambda_\sigma(i, i+1) = \lambda_\sigma(i-1, i) + d_b(i, \sigma) - d_a(i, \sigma), \ 1 \leq i \leq n-1 ,$$
$$\lambda_\sigma(0, 1) = \lambda_\sigma(n, n+1) = 0 .$$

The sum of areas of modules $m_{\sigma(1)}, m_{\sigma(2)}, \ldots, m_{\sigma(i)}$, is denoted by $\text{area}(i, \sigma)$, which is determined by the formula

$$\text{area}(i, \sigma) = \text{area}(i-1, \sigma) + \text{area}(m_{\sigma(i)}) ,$$

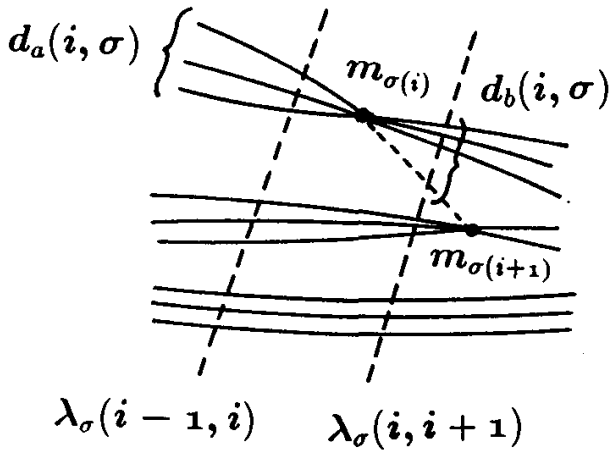


Fig. 5. The relation between $\lambda_\sigma(i-1, i)$ and $\lambda_\sigma(i, i+1)$.

where area$(0,\sigma) = 0$ and area$(n,\sigma) =$ area(S). Using area(i,σ) we can examine the area constraint in constant time.

Based on the above observations we implement a plane sweep on the arrangement of lines corresponding to given points to find an optimal partition. Since the abscissa of the dual plane corresponds to a slope of a line in the original plane, a plane sweep by a vertical sweep line from left to right corresponds to a circular (or angular) plane sweep like radar in the primal plane.

In the sweep on an arrangement of lines we move a vertical sweep line from left to right while maintaining a list (ordered sequence) of lines in the increasing order of the y-coordinates of their intersections with the sweep line, which is referred to as a y-list. The sweep starts from the negative infinity at which the y-list corresponds to the sorted list of lines in the increasing order of their slopes. For the initial y-list σ we compute $\min(N_j,\sigma)$ and $\max(N_j,\sigma)$ for each net N_j. Then, compute $d_a(i,\sigma)$ and $d_b(i,\sigma)$ for each $i, 1 \le i \le n$. Finally, $\lambda_\sigma(i,i+1), 1 \le i \le n-1$, are computed in the following manner.

$$\lambda_\sigma(0,1) = 0 \ ;$$

for $i = 1$ to $n-1$ do

$$\lambda_\sigma(i,i+1) = \lambda_\sigma(i-1,i) + d_b(i,\sigma) - d_a(i,\sigma) \ .$$

The above initialization can be done in $O(n \log n + M)$ time and $O(M+n)$ space.

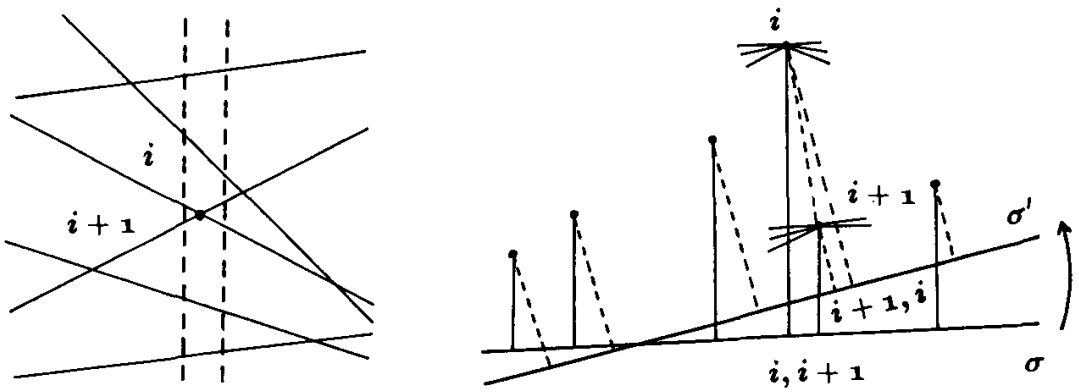
After the initialization we move the sweep line to the right. As is easily seen, when the sweep line passes over an intersection between lines, the order of lines, i.e., y-list changes. The next intersection can be computed in $O(\log n)$ time (refer to a standard text on computational geometry, for example [1] or [14]). Each time the sweep line encounters an intersection between two lines, we must exchange the order of those two lines in the y-list and update associated values such as $d_a(i,\sigma)$ and $\lambda_\sigma(i,i+1)$. This operation is called an elementary step.

Suppose that a current y-list is $(m_{\sigma(1)}, m_{\sigma(2)}, \ldots, m_{\sigma(n)})$ and the next intersection occurs between its ith and $(i+1)$st lines. Let the resulting y-list after the elementary step be σ'. Then, $d_a(\ )$ and $d_b(\ )$ change only for i and $i+1$. Referring to Fig. 6, we can easily see that

$$\begin{aligned} d_a(i,\sigma') &= d_a(i+1,\sigma) - e(\sigma(i),\sigma(i+1)) \ , \\ d_b(i,\sigma') &= d_b(i+1,\sigma) + e(\sigma(i),\sigma(i+1)) \ , \\ d_a(i+1,\sigma') &= d_a(i,\sigma) + e(\sigma(i),\sigma(i+1)) \ , \\ d_b(i+1,\sigma') &= d_b(i,\sigma) - e(\sigma(i),\sigma(i+1)) \ , \end{aligned}$$

where

$$e(\sigma(i),\sigma(i+1)) = \begin{cases} 1 & \text{if there is no such net } N_j \text{ that } N_j = \{m_{\sigma(i)}, m_{\sigma(i+1)}\} \ , \\ 0 & \text{otherwise.} \end{cases}$$



(a) Primal plane. (b) Dual plane.

Fig. 6. Elementary step.

Note that for any $j \neq i$ the value $\lambda_\sigma(j, j+1)$ remains unchanged, that is, $\lambda_{\sigma'}(j, j+1) = \lambda_\sigma(j, j+1)$. $\lambda_{\sigma'}(i, i+1)$ can be computed by

$$\begin{aligned}\lambda_{\sigma'}(i, i+1) &= \lambda_{\sigma'}(i-1, i) + d_b(i, \sigma') - d_a(i, \sigma') \\ &= \lambda_\sigma(i-1, i) + d_b(i+1, \sigma) - d_a(i+1, \sigma) + 2e(\sigma(i), \sigma(i+1)) \ .\end{aligned}$$

The same thing applies to area( ). That is, we have

$$\begin{aligned}&\text{area}(j, \sigma') = \text{area}(j, \sigma) \ \text{ for any } j \neq i, \text{ and} \\ &\text{area}(i, \sigma') = \text{area}(i-1, \sigma) + \text{area}(m_{\sigma(i+1)}) \ .\end{aligned}$$

All the above operation can be done in constant time if $e(\sigma(i), \sigma(i+1))$ can be retrieved in constant time. An obvious way to achieve constant-time retrieval is to use a two-dimensional array of size n^2. $O(n^2 + M)$ time suffices to determine all the elements of the array. However, using some hashing technique the time to build the data structure for constant-time retrieval can be reduced to $O(M)$. The tradeoff between retrieval time and space requirement will be discussed later.

When we sweep the entire arrangement of lines, elementary steps are performed $O(n^2)$ times. Thus, the total time required is $O(n^2 \log n + M)$. Here note that an elementary step itself can be done in constant time while $O(\log n)$ time is needed to find the next intersection.

Now, it may be obvious how to remove the constraint that no two nets coincide as a set. We have only to change the definition of $e(\sigma(i), \sigma(i+1))$ so that $e(\sigma(i), \sigma(i+1))$ represents the number of nets N_j such that $N_j = \{m_{\sigma(i)}, m_{\sigma(i+1)}\}$. Again, $O(M)$ time is enough to build the date structure.

Next, we shall describe how to remove the constraint that each net is a pin-pair. The key idea is to decompose each net into pin-pairs based on the convex hull of the point set corresponding to a net. That is, for each net we first compute its convex hull determined by the points to be interconnected by the net. Then, for

each adjacent points on the convex hull we create a pin-pair. In this way, a net consisting of k terminals is decomposed into at most $k+1$ pin-pairs. It should be noted that all the terminals are not associated with those pin-pairs, that is, connection to points which lie properly inside the convex hull is neglected (see Fig. 7). If a net is originally a pin-pair, we make one more copy of the net.

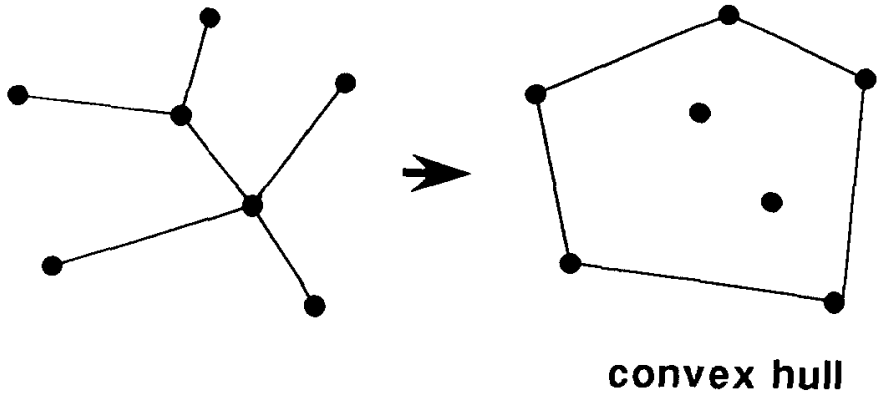


Fig. 7. Decomposition of a multi-terminal net into pin-pairs.

Now we have a simple but important observation.

Observation. When each net is decomposed into pin-pairs in the above-described manner, for each net a partitioning line intersects either none of its subnets or exactly two of them.

Thus, if we count the number of subnets and divide it by 2, we obtain the correct count.

Decomposition into pin-pairs is done as follows. First of all we sort the points in some order, say by their x-coordinates. Then, for each $m_{\sigma(i)} \in N_j$ we push j into stack[i]. If we repeat it for each net, we have a sorted list for each net. This is done in $O(n \log n + M)$ time. In practice we do not need to prepare many stacks. A simple one-dimensional array of size M together with m pointers are enough for the above calculation, where m is the number of nets. Since a sorted list is available for each net, the convex hull can be constructed in linear time. Thus, the total time required is $O(n \log n + M)$.

Recall that we needed $O(\log n)$ time only for finding the next intersection and all the remaining operations associated with an elementary step can be done in constant time. There is a standard technique to remove this $O(\log n)$ factor in an amortized sense, which is known as Topological Sweep [7]. In the plane sweep the sweep line is a straight line while a curved line is used in Topological Sweep which intersects every line exactly once.

The above results are summarized in the following theorem.

Theorem 1. *Given a set of n modules together with a net list of length M, we can find an optimal linear partition in the dual plane in $O(n^2 + M)$ time.*

3.4. Side-Fixed Modules

There may be several side-fixed modules in practical cases, that is, some modules (or pads) must be placed in predefined sides. For simplicity of the argument, let A be a set of modules that must be placed above a separating line and B be that of modules that must be placed below it. Then, we do not need to examine any cell above any line corresponding to a module in A or below any line corresponding to a module in B. So, the search space is restricted to the cells below any line of A and above any line of B, which results in the convex region bounded from above the lines of A and bounded from below the lines of B. If there are K intersections in the convex region consisting of s edges, Topological Walk [3] visits all the cells in the convex region in $O(K + (n + s)\log n)$ time. In our case s is smaller than n, and thus the overall time complexity is expressed as $O(K + M + n\log n)$.

3.5. Critical Nets

In practical cases some nets should be treated as *critical nets*, that is, those nets should be as shortest as possible with possible sacrifice of other nets. In such a case we should assign weights depending on their importance to those nets. Then, $d_a(i,\sigma), d_b(i,\sigma)$, and $\lambda_\sigma(i, i+1)$ must be defined not simply by the number of nets associated with but by the sum of weights of those associated nets. The algorithm remains essentially the same.

3.6. Time/Space Tradeoff

In the algorithm described above we need an efficient data structure to retrieve the value of $e(i,j)$, that is, whether there is a net interconnecting modules m_i and m_j. The simplest data structure mentioned above is a simple two-dimensional array of size n^2. Then, initialization of the array can be done in $O(M)$ time with constant retrieve time by a standard technique. One disadvantage of the date structure is its $O(n^2)$ space requirement.

Another candidate for this purpose is a balanced binary search tree. In the case, the space is $O(M)$ but the query time becomes $O(\log M)$. If we use $n^{(1/2)^k}$ tree (that means, a k—the level node has $n^{(1/2)^k}$ children), the retrieve time is decreased to $O(\log\log M)$ while the space increases to $O(nM^{1/2})$.

If we use $n^{(1/a)^k}$ tree for a positive constant a, then the space complexity becomes $O(M(n^2/M)^{1/a})$, which is $O(M^{1+\varepsilon})$ if a is sufficiently large. The time complexity becomes $O(a\log\log n)$. The $\log\log n$ factor can be removed by the following theorem.

Theorem 2. *Let S be a set of M elements of $\{1, 2, \ldots, N\}$. With $O(M(N/M)^\varepsilon)$ time and space, we have a data structure with $O(1)$ query time.*

Proof. Consider the following data structure: The root has M sons. Each internal node has $(NM)^\epsilon$ sons if it has at least one active (i.e., in S) descendant. Obviously, the depth of the tree is 1ϵ, which is $O(1)$. Since there are only M non-leaf nodes in each level of the tree, theorem follows. □

The result may not be good enough since the preprocessing time should be less than $O(M \log M/n)$, where $n = \sqrt{N}$. Next tradeoff is:

Theorem 3. *With $O(M + kM(N/M)^{1/N}2^{-k})$ time and space, we have a data structure with $O(k)$ query time.*

Proof. The data structure is given by letting $k = 1/\epsilon$. Further, for a node which has less than 2^k active descendants, we use usual binary search tree for searching in its active descendants. □

If we substitute $k = 2\sqrt{\log N/M}$, we have:

Theorem 4. *With $O(M)$ time and space, we have a data structure with $O(\sqrt{\log N/M})$ query time.*

Another approach is the one based on randomized algorithm. In deterministic sense, $O(\log \log M)$ query time hash is easily constructed in $O(M \log \log M)$ time by applying usual $\sqrt{n}$-ary search structure. The space complexity is $O(n\sqrt{M})$. If we apply randomization, $O(1)$ is possible. The easiest way is, first we randomize the sequence, and apply usual technique for designing a hash function.

Conclusions

In this paper we have presented an approximation algorithm for circuit partitioning problem. In the algorithm modules are mapped into points in the plane so that those points corresponding to two modules tightly interconnected are laid closely. Then we enumerate all possible linear partitions of the point set. The presented algorithm finds a best partition among them in time $O(n^2 + M)$ where n is the number of modules and M is the total length of a net list. Since there are $O(n^2)$ different linear partitions, the nonredundancy of the algorithm is easily seen. Of course, the effectiveness of the approach heavily depends on the transformation of modules into points in the plane. In this paper we are not so interested in the problem but in efficient examination of partitions.

Acknowledgments

The first author would like to thank Dr. R. S. Tsay at IBM Watson Research Center for introducing the problem. The authors would also like to thank Dr. J. Matousek for his valuable discussion. Finally the authors would like to express their

gratitude for a reviewer of the original manuscript for his valuable suggestions and giving a number of references.

References

1. T. Asano, "Computational geometry", *Asakura-shoten*, (in Japanese) 1990.
2. T. Asano, B. Bhattacharya, J. M. Keil and F. F. Yao, "Clustering algorithms based on minimum and maximum spanning trees", *Proc. 4th Ann. ACM Symp. Computational Geometry*, (Urbana-Champaign, 1988) pp. 252–257.
3. T. Asano, L. J. Guibas and T. Tokuyama, "Walking on an arrangement topologically", *Proc. 7th Ann. ACM Symp. Computational Geometry*, (North Conway, 1991) pp. 297–306.
4. B. Chazelle, L. J. Guibas and D. T. Lee, "The power of geometric duality", *Proc. 24th Annual IEEE Symp. on Foundations of Computer Science*, (Tucson, 1983) pp. 217–225.
5. W. E. Donath, "Logic partitioning", in *Physical Design Automation of VLSI Systems*, eds. B. Preas and M. Lorenzetti, (Benjamin/Cummings, 1988), pp. 65–86.
6. W. E. Donath and A. J. Hoffman, "Lowere bounds for the partitioning of graphs", *IBM J. Res. Dev.*, 1973, pp. 420–425.
7. H. Edelsbrunner and L. J. Guibas, "Topologically sweeping an arrangement", *Proc. 18th Annual ACM Symposium on Theory of Computing*, 1986, pp. 389–403.
8. J. Frankle and R. M. Karp, "Circuit placement and costs bounds by eigenvector decomposition", *IEEE Intl. Conf. on Computer-Aided Design*, 1986, pp. 414–417.
9. M. R. Garey and D. S. Johnson, "Computers and intractability", (Freeman, New York, 1979).
10. M. R. Garey, D. S. Johnson and L. Stockmeyer, "Some simplified NP-complete graph problems", *Theor. Comput. Sci.* 1 (1976) pp. 237–267.
11. K. M. Hall, "An r-dimensional quadratic placement algorithm", *Management Science*, 17 (1970) 219–229.
12. U. Lauther, "A min-cut placement algorithm for general cells assemblies based on a graph representation", *J. Digital Systems*, 4 (1980) 21–34.
13. R. H. J. M. Otten, "Automatic floorplan design", *Proc. 19th Design Automation Conf.*, 1982, pp. 261–267.
14. F. P. Preparata and M. I. Shamos, "Computational geometry — An introduction", (Springer-Verlag, New York, 1985).
15. G. Sigl, K. Doll and F. M. Johannes, "Analytical placement: A linear or quadratic objective function?", *Proc. ACM/IEEE Design Automation Conf.*, June 1991, pp. 427–432.
16. R. S. Tsay and E. S. Kuh, "A unified approach to partitioning and placement", *Princeton Conf. on Inf. and Comp.*, 1986.

Algorithmic Aspects of VLSI Layout
Eds. M. Sarrafzadeh and D. T. Lee
© World Scientific Publishing Company

THE THREE-DIMENSIONAL CHANNEL ROUTING PROBLEM

MARTIN L. BRADY
Department of Computer Science and Engineering, The Pennsylvania State University, 231 Pond Lab., University Park, PA 16802, USA

DONNA J. BROWN
Coordinated Science Laboratory, University of Illinois, 1308 W. Main St., Urbana, IL 61801, USA

and

PATRICK J. McGUINESS
Semiconductor Systems Design Technology, Motorola Inc., Mail Drop OE 321, 6501 William Cannon Drive W., Austin, TX 78735-8598, USA

ABSTRACT

This paper presents a near-optimal routing algorithm for the three-dimensional channel routing problem, or 3dCRP. The 3dCRP is an extension of standard channel routing models, which allows not only multiple routing layers, but also each column may on a shore contain multiple terminals stacked one above another in different layers. The main algorithm presented operates in three phases to route two-terminal nets, and uses at most $d/(L-p) + O(p\sqrt{d(L-p)})$ tracks, where L is the number of layers, p is the number of terminals per stack, and d is the problem density. We also describe an algorithm for the multiterminal net 3dCRP, that uses at most $2d/(L-p) + O(p(L-p)\sqrt{d})$ tracks.

Keywords: Channel routing, multilayer routing, VLSI layout, density, wire overlap.

1. Introduction

The problem of automatic wire routing is extremely important in the layout of integrated circuits. The vast majority of research in VLSI wire routing has centered on two-layer wiring models, treating the problem on a two dimensional grid. However, as more layers of interconnect become feasible, it becomes necessary to consider wire routing as a truly three-dimensional problem. In this paper we address the *Three-Dimensional Channel Routing Problem* (3dCRP), a generalization of the familiar VLSI channel routing problem.

Standard two-layer channel routing models assume that two opposite sides, or shores, of a rectangular channel contain sets of terminals, which are to be joined by wires. Terminals are assumed to occupy both layers on the shore, so only one terminal per channel gridpoint is allowed. Wire routing proceeds by partitioning the general wiring problem into a set of channels and sequentially solving the resulting

CRPs. Our three-dimensional CRP model assumes some constant number of layers, L, are available for routing, and each layer of the shore may contain a different terminal; that is, each shore gridpoint may have up to L terminals. Some previous work has considered multilayer channel routing.[1–3] However, while these models assume L layers, they restrict terminals to one per shore gridpoint. Our 3dCRP model is very general, and should cover a variety of routing situations. Besides standard multilayer VLSI chips, the results are applicable to routing PC boards, which could contain dozens of layers, and to multi-chip modules, consisting of silicon or ceramic substrates with perhaps six routing layers.[4,5]

Let us more precisely define the 3dCRP problem. The channel is a three-dimensional grid consisting of planes called *tracks* (or rows), *columns*, and *layers*. Fig. 1 gives a view of the three-dimensional channel. The top and bottom tracks of the channel, 0 and $w + 1$, are the *shores*. The *width* of a channel, w, is the number of tracks it contains (excluding the shores), the number of columns is n, and the number of layers is L. The intersection of a column and a track consists of L gridpoints which we refer to as a *stack*. The intersection of a track and a layer is a *lane*. Any gridpoint on either shore can be labelled with a number, which corresponds to a *terminal* assignment, and terminals with the same number form a *net*. The set of terminals for each net must be connected through the grid in such a way that each net's wiring uses a disjoint path. Unlabelled shore gridpoints are assigned to no net; we refer to these as *empty terminals*. A net with q terminals is a *q-terminal net*, and if $q > 2$ it is a *multiterminal net*. We refer to a two-terminal net which has both terminals in the same column (although not necessarily the same layer) on opposite shores as a *trivial net*. For reasons we shall explain in Sec. 2, we restrict the number of terminals (or pins) allowed in any stack to a maximum of $p < L$. The goal is to minimize the width of the channel (i.e., the number of tracks). We distinguish between *horizontal routing* (routing between columns) and *vertical routing* (routing between layers). For channel routing problems, it is reasonable to assume that the number of layers is constant and small relative to the number of columns. The horizontal routing problem is then the main issue, as we will show that vertical routing can be done in a number of tracks which is a function only of the number of layers.

Our best algorithms use a multi-phase approach, first proposed by Baker et. al.[6] for two-layer routing, and used by Berger et. al.[2] for restricted multilayer channel routing, where there were L layers and $p = 1$. The channel is partitioned into *blocks* of r consecutive columns each. The basic strategy is to route the nets in three hierarchical phases. The first two phases perform the interblock routing: each net is routed from the block in which it begins to the block in which it ends – but not necessarily to the correct column or layer within the block. Phase 3 then performs the intrablock routing, where each net is routed to its precise column and layer terminal location. Fig. 2 provides an overview of the overall routing algorithm, illustrating the phases and the number of tracks used in each phase.

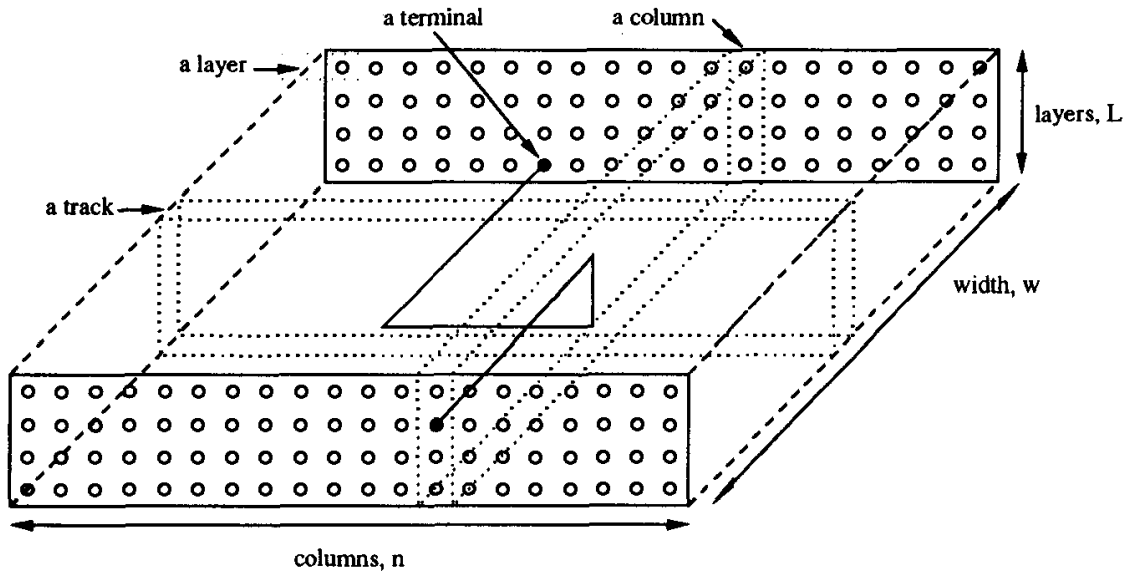


Fig. 1. The three-dimensional channel.

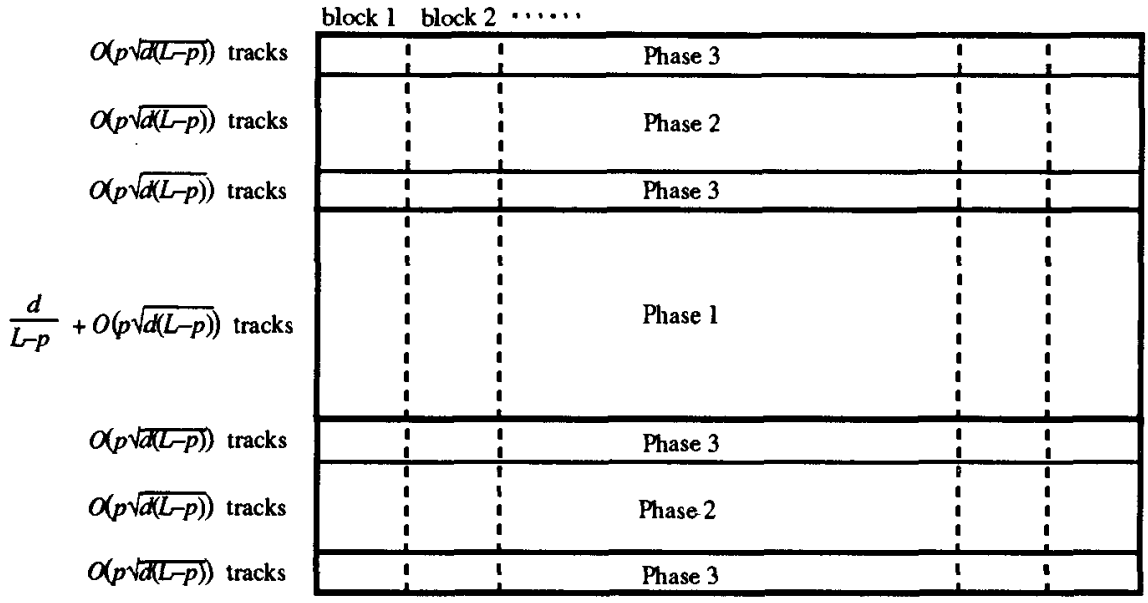


Fig. 2. Overview of the three-phase 3dCRP algorithm.

In Sec. 2 we observe some simple lower bounds for the 3dCRP, which give parameters for estimating the constraints necessary for reasonable routings. Section 3 gives a simple but general algorithm for three-dimensional channel routing for $p < L$. In Sec. 4 we present our improved but more complex three-phase algorithm for two-terminal net problems, which uses the Sec. 3 algorithm as the third phase. The width of the channel is nearly optimal: $d/(L-p) + O(p\sqrt{d(L-p)})$ tracks. Section 5 discusses the generalization of our algorithm to handle multiterminal nets, leading to an algorithm which routes multiterminal net 3dCRPs

in $2d/(L-p) + O(p(L-p)\sqrt{d})$ tracks. Section 6 contains some comments and conclusions.

2. Lower Bounds

In this section we give some lower bounds for the 3dCRP. Density provides a fundamental lower bound for standard two-layer CRPs, and is similarly defined for the 3dCRP. The density of a column is the number of nets which have a terminal on both sides of the column, and therefore must cross the column. The maximum over all columns is d, the *density* of the problem. Obviously, d lanes will be needed for any problem, so $w \geq d/L$ tracks are necessary. This bound can be improved by observing that trivial nets routed directly across the channel add no density to the problem, but use up layers that might have otherwise been used for horizontal wires crossing dense cuts. Using this idea, it has been shown that for $p = 1$, a lower bound of $w \geq d/(L-1)$ can be obtained,[7] and generalized for any $p < L$ to $w \geq d/(L-p)$.[8]

Our results all assume that there exist at most $L-1$ terminals in a stack, i.e., $p < L$; indeed the bounds that we obtain are possible only when there are a sufficient number of empty terminals. In particular, let e be the number of empty terminals on a shore. If we allow $p = L$ (in too many stacks) then strong lower bounds can be obtained that are not dependent on the density, but instead depend on a function of n, whenever $e = o(n)$. To route a wire horizontally on track i, we must ensure that the lane segment in which the wire routes is free of other nets; the lane must have had its wires vacated in prior tracks. Thus, if a net is surrounded by wires "on all sides" (i.e., in neighboring columns and on adjacent layers in the same column) that net cannot move until one of its neighboring nets moves.

Consider the extreme case where every possible terminal location is used, that is, $e = 0$. Assuming the nets are nontrivial, it is impossible to even route the problem without going outside the channel. Applying the argument of Brown and Rivest,[9] we observe that the only nets routable in the first track are those in columns 1 and n. Proceeding inductively, nets in columns j and $n-j+1$ cannot have their wires moved until at least track j. Thus, column $n/2$ cannot be moved until track $n/2$ so, assuming there are no trivial nets in the problem instance, at least $n/2$ tracks are required for the problem.

The same argument can be extended to the case where $0 < e < n$. There must be some column i in the channel that is at least $\frac{(n-e)}{2(e+1)}$ columns away from a column with an empty terminal (or from a column outside the channel). This implies that column i cannot be routed until track $\frac{(n-e)}{2(e+1)}$. Hence, if there are no trivial nets, then the channel width is at least $\frac{(n-e)}{2(e+1)}$. These bounds use very simple counting arguments and can certainly be improved by more careful examination. Nevertheless, they show that if the number of unused terminals becomes too small, the channel rapidly becomes very wide. And if e is at most a sublinear function of n, then the channel width is lower bounded by a function of the number of columns,

independent of the density of the problem. Thus, we restrict our attention to problems in which $p \leq L - 1$.

3. Intrablock Routing Algorithm

In this section we describe our Intrablock Routing Algorithm, which presents a straightforward solution to the 3dCRP. Although its performance may in general be quite poor, it has the advantage of being simple and, moreover, it performs well as Phase 3 of our main algorithm.

As mentioned previously, we view our routing as being partitioned into blocks of r consecutive columns, labeled from left to right as Block 1, Block 2, ..., Block $\lceil n/r \rceil$. For the Intrablock Routing Algorithm, we assume that each net has all its terminals in a single block. This will turn out to be the case at the end of Phase 2 in our Three-Phase Algorithm. However, note that *any* 3dCRP problem can be viewed as consisting of a single block of size $r = n$. We shall show that $O(rp)$ tracks are sufficient to route any 3dCRP, where r is the block size and $r > p$. Thus, the smaller the block size, the better the performance.

Within each block, the algorithm can perform any net routing desired. The routing proceeds track-by-track (in contrast to the column-by-column method that will be used in Sec. 4), and "up" and "down" signify movement between layers. Since $p < L$, each shore stack has at least one empty terminal, and each shore block has at least r empty terminals. Without loss of generality, we assume that in each shore stack the p terminals occupy layers $1, \ldots, p$. If not, then at most p tracks are needed to move them down into the first p layers.

The Intrablock Routing Algorithm routes the nets into their correct positions one layer at a time, beginning with layer p, and iterating down to layer 1. For concreteness in describing the algorithm, we shall assume in this section that $p = L-1$. If $p < L-1$, then the problem is strictly easier because more space is available within each block, and fewer than $L - 1$ passes are needed.

In order to have room to reposition the nets within the block, we borrow an additional $r+p$ empty spaces from adjacent blocks: p from the block to the left and r from the block to the right. For this reason, it is not possible to route blocks $B-1$ or $B+1$ while we are routing block B; thus, the algorithm routes only one-third of the blocks at a time. Specifically, our Intrablock Routing Algorithm first completely routes all blocks i such that $i \equiv 0 \bmod 3$ and within these blocks it routes layer $L-1$, then layer $L-2$, etc., down to layer 1. Then the algorithm proceeds to route the next third of the blocks (all blocks i such that $i \equiv 1 \bmod 3$) and finally the last third of the blocks (the blocks i such that $i \equiv 2 \bmod 3$).

In Sec. 3.1, we detail the individual steps of the Intrablock Routing Algorithm for two-terminal net 3dCRPs, including an analysis of the number of tracks used. Section 3.2 provides pseudo-code for the INTRABLOCK(B, $L-1$) procedure, which implements one pass of the algorithm. Finally, Sec. 3.3 describes the extension of the Intrablock Routing Algorithm to multiterminal nets.

3.1. Routing Two-Terminal Nets

The general strategy is to route one layer at a time, from layer p down to layer 1. To route block B, some empty space must first be claimed in blocks $B-1$ and $B+1$. Then, layer p is routed by moving nets with destinations in layer p into empty space in block $B+1$, vacating layer p in block B, and finally routing the nets into the vacated layer. This is repeated for $p-1$, and so forth (the first step, claiming space in $B-1$ and $B+1$, need not be repeated).

We describe the Intrablock Routing Algorithm's operation on a single block B, as it reorders a single layer, $L-1$ (assuming the worst case, that $p = L-1$). Those nets which need to be routed in the current pass (the nets that have destination terminals in layer $L-1$) are the *selected nets.* Layer L is initially empty and so is available for routing within the block; that is, it is the current *routing layer.*

Step 1. Make additional empty space available in blocks $B-1$ and $B+1$.

Using p tracks, the nets in the rightmost stack of block $B-1$ (closest to block B) are moved up to layer L (occupying, for example, columns $r-p, \ldots, r-1$ of block $B-1$), thereby forming an empty stack which we call the *free stack.* Also, layer L of block $B+1$ is reserved for block B; these r empty spaces are the *temporary positions* (see Fig. 3(a)).

Step 2. Move selected nets in B into temporary positions in layer L of block $B+1$.

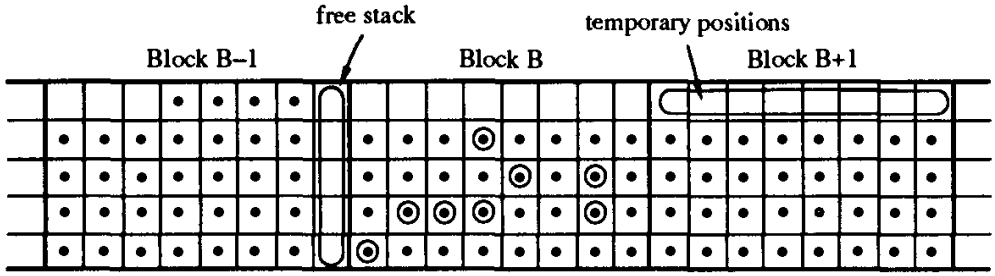
There are (at most) r nets in block B that need to move to layer $L-1$. We examine the stack of nets adjacent to the free stack (initially the leftmost stack of B). If any selected net is found in this stack, that net is pulled out of its stack, routed up the free stack to layer L, and then moved into the rightmost available temporary location in block $B+1$. This routing takes only r tracks, one for each selected net (see Fig. 3(b)). After all the selected nets in a given stack have been routed to the temporary positions, the free stack is moved right, and the removal of selected nets is repeated for the next stack. (The empty free stack is "moved" right, using a single track: move one column left every net that is in the stack to the right of the free stack.) The selection procedure continues until all r stacks in block B have been examined and all r selected nets are in the temporary positions in block $B+1$.

Step 3. Move the empty positions (vacated by selected nets in (2)) into layer $L-1$.

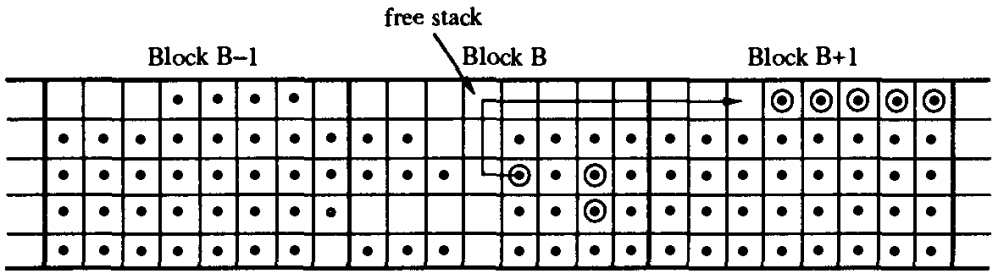
Since the selected nets have been removed, block B now contains at most r "holes" where the selected nets used to be. Within each stack in B, we move all nets to the lowest set of layers possible, in effect moving the holes to the top layers; this takes at most $p-1$ tracks. In no more than r tracks, the (at most) r nets in layer $L-1$ of block B are moved in order to fill in holes in lower layers in B, thereby vacating layer $L-1$ in block B.

Step 4. Move selected nets to final destinations.

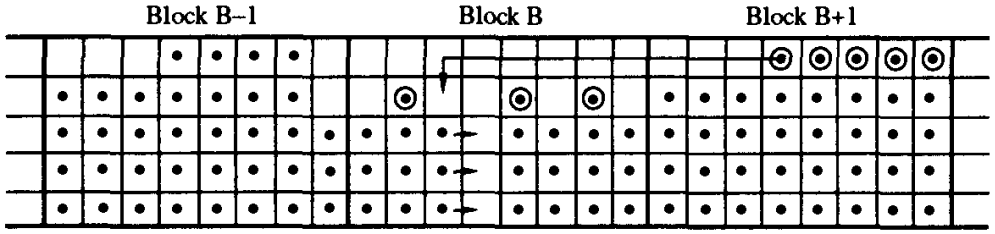
All of the r selected nets destined for layer $L-1$ are moved from the temporary locations back to layer $L-1$ of block B. Moving one net per



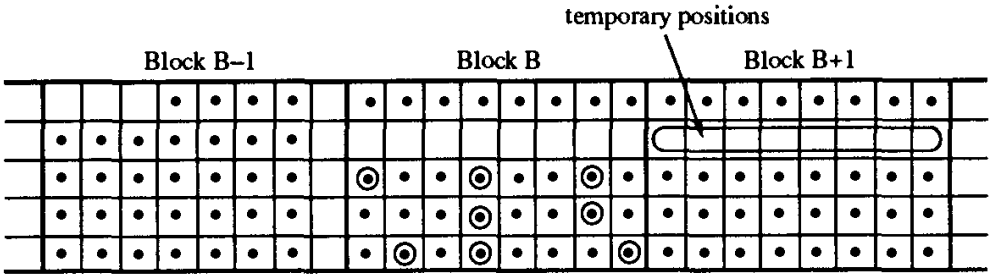
(a) Step 1: Reserve the free stack and temporary positions.



(b) Step 2: Move selected nets into temporary positions.



(c) Step 4: Move selected nets to final destinations.



(d) Block B just prior to routing layer $L - 2$. New selected nets are marked.

Fig. 3. The Intrablock Routing Algorithm: Routing layer $L - 1$ in block B ($L = 5$, $r = 8$). • denotes a net and ⊙ marks selected nets.

track, we individually route each net over and down to its precise destination terminal in layer $L-1$. (Note that the r selected nets can be permuted into any order as they are placed in layer $L - 1$.) Within the same r tracks, the free stack is propagated back to its starting place, the rightmost column of block $B - 1$ (see Fig. 3(c)).

Having routed layer $L-1$, the algorithm next routes layer $L-2$ in block B. To do this, the nets occupying layer $L-1$ in blocks B and $B+1$ are moved up to layer L so that layer $L-1$ is available for the temporary positions and for routing. The selected nets are now the nets with destinations in layer $L-2$. Fig. 3(d) illustrates the net positions at the start of this second pass; notice that the free stack is now back in its starting position, the rightmost column of block $B-1$. Step 1 can therefore be omitted for all remaining layers. (In addition, the needed stack height decreases by 1 with each pass.) This process is repeated until all layers in block B have been routed. Finally, the nets in B must all be moved down to layers 1 through p and, simultaneously, the nets moved in block $B-1$ to form the free stack are returned to their original locations. This algorithm is summarized below.

```
for i = 0 to 2
{   for all blocks B ≡ i mod 3
    {   Step 1. Make additional empty space available in blocks B − 1 and B + 1.
        for l = p to 1 step −1
        {   Step 2. Move selected nets in B into temporary positions in layer
                    l + 1 of block B + 1.
            Step 3. Move the empty positions into layer l.
            Step 4. Move selected nets to final destinations.
        }
    }
}
```

We now total up the number of tracks used. Forming the free stack uses p tracks, and Steps 2 through 4 use $r+p-2$ tracks: $2r-1$ tracks to propagate the free stack across the block and to route the selected nets to the temporary locations, $r+p-1$ tracks to move and "level out" the holes for the destination layer, and r tracks to (simultaneously) place the selected wires and return the free stack to its original position. In one track the layer just routed is moved up one layer, and we are ready to begin the next pass. Repeating for the next layers, and then returning all nets to their original locations uses a total of

$$p + (4r + p - 2)p + (p-1) + p \quad \text{or} \quad 4rp + p^2 + p - 1$$

tracks, to route one third of the blocks. Repeating for the next third of the blocks and then the next, the algorithm described uses $3(4rp + p^2 + p - 1)$ tracks. Since $p < r$, this is fewer than $15rp$ tracks. This yields the track usage bound for the algorithm.

Theorem 1. *Given a two-terminal net 3dCRP that consists of blocks of size r such that $p < r$ and $p < L$, the Intrablock Routing Algorithm routes the nets using at most $15rp$ tracks.*

3.2. Procedure INTRABLOCK(B,L − 1)

The four steps described in Sec. 3.1 are written more formally below in the pseudo-code procedure INTRABLOCK(B, $L-1$), which describes the operation of the algorithm within block B as it specifies the detailed routing of nets destined for layer $L-1$. We first need some notation in order to formally describe the algorithm. A column/layer position in a given track is described by giving its block, column within the block (between 1 and r), and layer number. Thus, (B, c, l) denotes a wire at the cth column of block B (i.e., column $(B-1)r+c$ overall) in layer l. (Occasionally, we allow c to be outside the range 1 through r; this simply denotes a column outside the specified block. For instance, (B, 0, l) would refer to the column immediately left of (B, 1, l): that is ($B-1$, r, l).)

The notation ROUTE(t) is used to introduce routing between columns and layers of a wire in track t. For instance, ROUTE(t): $(B, 1, 1) \Rightarrow (B, 1, L) \Rightarrow (B, 2, L)$ indicates routing the wire at column 1, layer 1 in block B first up to layer L and then over into column 2, all in track t. Between two tracks, t and $t+1$, all wire end points are extended from track t to $t+1$. If no net exists at a starting point specified by ROUTE, the routing is simply ignored. A few additional shorthand terms are used. NET(B, c, l) returns the net number of the net currently at position (B, c, l) in the current track t. NET(B, c, l) = 0 implies that no net currently occupies position (B, c, l) (i.e., an empty position). COL(i) returns the destination column number (within the proper block) of net number i. Finally, MIN_EMPTY is used to obtain the leftmost currently empty position within a specified range, and among those in the same column, the one of lowest layer number. (The free stack is excluded from consideration by MIN_EMPTY.) For example, MIN_EMPTY ((B, c), (B, $c+2$)) returns the position (block, column, and layer) of the leftmost/lowest empty position between columns c and $c+2$ (inclusive) in block B.

procedure INTRABLOCK(B, $L-1$)
Step 1. Create the free stack.
$t = 1$
for $i = L-1$ to 1 **step** -1
{ ROUTE(t): $(B-1, r, i) \Rightarrow (B-1, r, L) \Rightarrow (B-1, r-i, L)$
$t = t+1$
}
Step 2. Move selected nets to layer L of block B + 1, and move the free stack right.
$s = 0$
for $i = 1$ to r
{ **for** $j = 1$ to p
{ **if** NET(B, i, j) is selected

```
            { ROUTE(t): (B, i, j) ⇒ (B, i − 1, j) ⇒ (B, i − 1, L)
                        ⇒ (B + 1, r − s, L)
                        s = s + 1
                        t = t + 1
            }
    }
    if i ≠ r
    {       for j = 1 to L − 2
            { ROUTE(t): (B, i, j) ⇒ (B, i − 1, j)  }
            t = t + 1
    }
}
```

Step 3. Shift nets down into the lowest layers, then vacate layer $L-1$.

```
for k = 1 to L − 2
{   for i = 1 to r
    {   for j = L − 2 to 1 step −1
        {   if NET(B, i, j) = 0
            {   ROUTE(t): (B, i, j + 1) ⇒ (B, i, j)   }
        }
    }
    t = t + 1
}
for i = 1 to r
{   (β, γ, λ) = MIN_EMPTY((B − 1, r), (B, r))
    if NET(B, i, L − 1) ≠ 0
    {   ROUTE(t): (B, i, L − 1) ⇒ (β, γ, L − 1) ⇒ (β, γ, λ)
        t = t + 1
    }
}
```

Step 4. Move selected nets to final destinations, and move the free stack left.

```
for i = 1 to r
{   ROUTE(t): (B + 1, i, L) ⇒ (B, COL(NET(B + 1, i, L)), L) ⇒
                              (B, COL(NET(B + 1, i, L)), L − 1)
    for j = 1 to L − 2
    {   ROUTE(t): (B, r − i, j) ⇒ (B, r − i + 1, j)  }
    t = t + 1
}
```

3.3. Extension to Multiterminal Nets

In the extension of the 3dCRP algorithm to multiterminal nets, it is necessary to merge together, in the top and bottom tracks, the terminals in each block that belong to the same net. This ensures that the interblock routing has at most one

terminal belonging to a net on each side of the channel in any given block. We extend the Intrablock Routing Algorithm, modifying the algorithm so that terminals in the same block belonging to the same net are merged together during the intrablock routing.

The terminal merging operation is done in Step 4 of the routing iteration: In Step 4, the r selected nets in the temporary locations are moved to the destination layer of the block; each net is individually moved, one net per track, over and down to its precise destination terminal. If more than one net has the same column and layer destination, subsequent nets are merged with the first net as they are routed into the destination. (If two selected net branches are to be merged, they have the same destination column and layer.)

The only further difficulty is that there may be more than r terminals destined for a single layer. (For example, suppose two terminals are destined for each terminal location in a layer.) The extended algorithm repeats the basic layer operation on each set of r terminals destined for that layer until all the terminals with destination in that layer are routed. Steps 2 and 4 are repeated, but Step 3 is omitted, as we need not clear any additional nets from the destination layer. There are at most rp terminals per block, so that there are at most $p-1$ additional sets of nets beyond the first set in any layer. In the worst case, consider $p-1$ layers that have only one net destined for that layer, and finally a single layer that has $(r-1)p+1$ nets destined for it. Thus, there are at most $2p-1$ iterations of the layer reordering operation needed to route and merge the nets in a single block (i.e., $p-1$ "extra" passes to complete the single heavily used layer). Each extra pass requires $3r-1$ tracks, and the entire procedure is repeated three times, so at most $3(3r-1)(p-1) < 9rp$ additional tracks are used for multiterminal net routing.

Theorem 2. *Given a multiterminal net 3dCRP that consists of blocks of size r such that $p < r$ and $p < L$, the Intrablock Routing Algorithm routes and merges the nets using at most $24rp$ tracks.*

4. A Three-Phase Algorithm for the Two-Terminal Net 3dCRP

In this section we describe a 3dCRP algorithm for two-terminal net problems. We assume the two terminals of each net are located on opposite shores. In each stack on either shore of the channel, there are at least $L-p$ empty terminals. We define the constant $k = L-p$ for convenience. We assume the non-empty terminals are in the lowest layers $1, \ldots, p$. (If the terminals were arbitrarily distributed in a stack, they could be moved to layers $1, \ldots, p$ using at most p tracks.) The algorithm uses at most $d/2$ additional columns outside (to the left of) the channel. Our three-phase algorithm routes the two-terminal net 3dCRP in $d/(L-p)+O(p\sqrt{d(L-p)})$ tracks. For the case with the most densely packed terminals, $p = L-1$, the bound is $d+O(L\sqrt{d})$ tracks. We assume that p and L are fixed and small relative to d, and thus the major challenge in the algorithm is horizontal routing. In particular, these bounds require that $d > Lp$.

Our algorithm extends the two-phase approach used in Berger et. al.[2] for multilayer (i.e., L layers, $p = 1$) channel routing. The channel is partitioned into *blocks* of r consecutive columns each. Our basic strategy is to construct the paths in three hierarchical phases, as described below. Recall Fig. 2, which depicts the partitioning of the channel into phases and blocks.

Phase 1. Approximate Interblock Routing.

The first phase performs an approximate routing of long wires to positions near their terminals. Tracks in the middle section of the channel are used to route each net from the block containing its leftmost terminal to the block containing its rightmost terminal, *or to a block as close to it as possible.* In any case, the density of the remaining top and bottom subproblems is at most $O(rp)$.

Phase 2. Exact Interblock Routing.

The second phase completes the interblock routing, in the top and bottom sections of the channel, so that each net is routed to its correct block. This is accomplished using a special case of the Phase 1 algorithm.

Phase 3. Intrablock Routing.

Finally, each net is routed to its correct terminal location (column and layer) within each block, in each of the four remaining bands separating Phase 1, Phase 2, and the upper and lower shores of terminals. This is accomplished using the Intrablock Routing Algorithm described in the previous section.

4.1. Phase 1: Approximate Interblock Routing

We first describe the approximate routing technique which is the heart of the three-phase algorithm. In Fig. 2, the channel is viewed from above: tracks appear as horizontal lines, columns as vertical lines, and layers are stacked on one another. We use this orientation for top, bottom, horizontal, and vertical in the description of Phase 1; note that "up" and "down" refer to movement toward the top and bottom shores, respectively. We begin with a few more definitions. A *rising* (*falling*) *net* has its rightmost terminal on the top (bottom) shore of the channel and in a different block than its leftmost terminal. A *vertical net* has both its terminals in the same block. A starting (*ending*) *net* in a block B is a net which has its leftmost (rightmost) terminal in B. A *continuing net* in B has terminals in blocks both to the left and to the right of block B. A net is *extended* in block B if its rightmost terminal is in a previous block but the net was unable to exit and is still in the channel. Fig. 4 shows examples of some of the defined types of nets: nets 1 and 2 are falling nets, net 3 is a vertical net, and net 4 is rising, net 1 is ending in block i, net 2 is a starting net in block i.

The Phase 1 algorithm consists of three subphases: Phases 1.0, 1.1, and 1.2. Phase 1.0 is used to modify the routing problem slightly in order to facilitate the next subphase. It turns out to be advantageous to have the number of ending nets

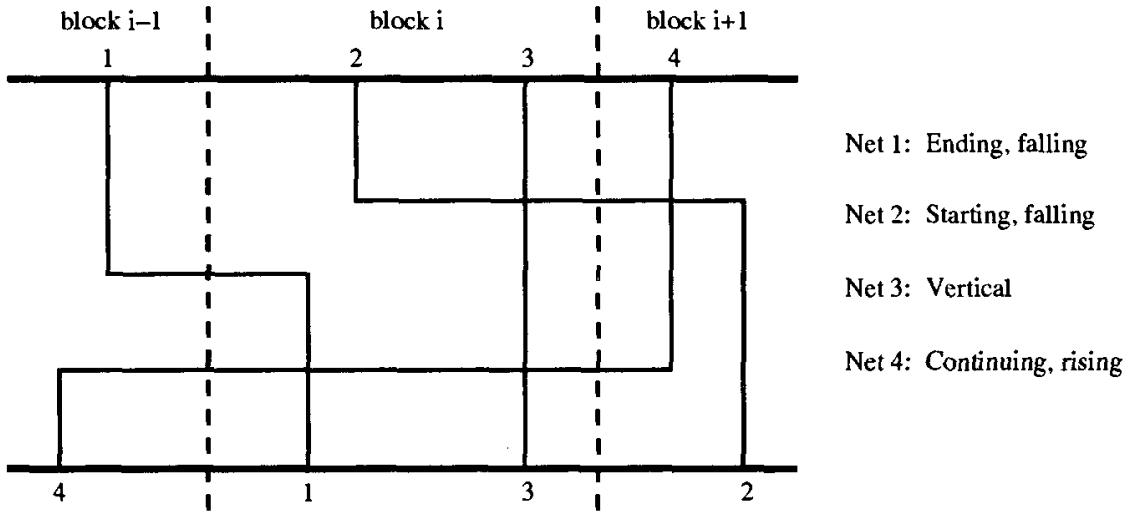


Fig. 4. Illustration of various net terminologies with respect to block i.

within a block be a multiple of $p(L-p)$, so that we never need to route both ending and starting nets in the same column. Phase 1.0 is used to swap ending nets for starting nets between blocks to ensure this condition.

Phase 1.1 proceeds column by column and block by block from left to right. Within each block, ending nets are routed in the leftmost columns of the block, followed by the starting nets, and finally vertical nets. This ordering of nets ensures an easier routing strategy without increasing the density of the routing problem. The strategy is to pack the falling nets into the lower tracks of the channel and the rising nets into the upper tracks of the channel as soon as they enter the channel. Rising and falling ending nets can then be routed out of the channel without conflict using the leftmost columns of the block. The $L - p$ upper layers are used for horizontal routing across tracks, while layers 1 through p are used for routing nets vertically in the columns.

Phase 1.1 leaves a number of unused lanes within the rising and falling regions because of necessary gaps created when ending nets are routed out of the channel and rising and falling nets are routed around one another. We refer to such lanes as *bad lanes*, and to a track containing at least one such empty lane as a *bad track*. In Phase 1.2, the bad lanes are reclaimed by moving empty lanes back into the center of the channel, where they are again available for horizontal routing, and are therefore called *good lanes*.

Another difficulty is the routing of wires between the vertical and horizontal routing layers when they change directions. The p layers $1, \ldots, p$ are reserved for vertical routing, and the remaining $L - p$ layers, $p + 1, \ldots, L$ are used for horizontal routing. Thus, at the intersection of any given column and track, only one net can move between the horizontal and vertical routing layers and we call such a net *exposed*. This creates a bottleneck in the movement of nets from vertical to horizontal and vice-versa. For example, if a net is an ending net in a horizontal

lane in the current block, it may exit only if it can make a via to a vertical routing layer. If there is a non-ending net in the same track on a lower layer, then the ending net is unable to route to a vertical layer and exit. We say an ending net is *positioned* to exit if it is in a horizontal lane that is not blocked by a non-ending net; a net is *non-positioned* if it is blocked. A track is *properly ordered* if all its ending nets are positioned. A second function of Phase 1.2 is to position the nets in each track into their proper order. All of the routing in Phase 1.2 occurs concurrently with Phase 1.1, and does not interfere with the Phase 1.1 routing of nets.

Before describing the details of each subphase of Phase 1, we discuss some obstacles to be overcome. A major problem occurs in making connections between the horizontal and vertical layers. Any nets which want to exit but are non-positioned become extended nets in subsequent blocks, until a block is reached in which they are finally able to exit. Care must be taken to limit the number of extended nets which can occur, in order to limit the density of the remaining subproblems. An additional aspect of the problem is how to efficiently make connections between vertical and horizontal routing layers, when many wires in the same track or column need to change direction. In our strategy, a single column of p nets is split into p tracks to turn horizontally; likewise, a single track of $k = L - p$ horizontal nets is split into k columns to route vertically. We sometimes refer to a set of k contiguous columns as a *segment*, and a set of p contiguous tracks as a *group* of tracks. In general, bends in the routing are organized so that a set of pk nets in k contiguous columns (a segment) are routed into the horizontal layers and leave the segment in a group of p tracks, as shown in Fig. 5. Consequently, staircase-shaped patterns of continuing and starting nets that are formed consist of *steps*, k columns wide and p tracks high, each step containing pk nets which change direction.

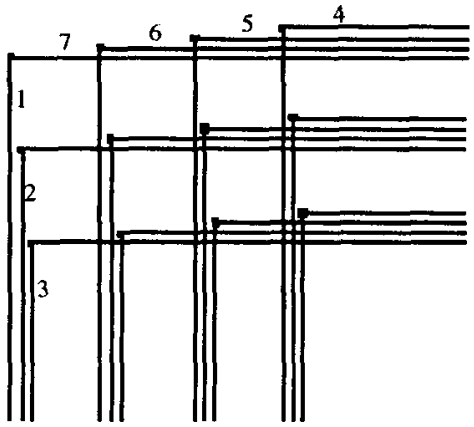


Fig. 5. Illustration of a staircase step of pk bending wires for the case $L = 7$, $p = 3$.

Since nets cross the channel as soon as they enter, rising and falling starting nets have to be routed around each other. To do this, we maintain a *pyramid structure* of unused track and column sections from previous columns that are available for

routing starting nets. Each pyramid path is a C-shaped path consisting of track sections on each side of the channel and joined by a column section. The pyramid structure is initialized using $d/2p$ columns to the left of the channel. Rising and falling nets are routed around each other by routing one side directly and routing the other side using pyramid paths. For every pair of ending nets, the pyramid expands by one path, while every pair of starting nets consumes one pyramid path (see Baker et. al.[6] for an introduction to the use of the routing pyramid).

4.1.0. Phase 1.0: problem modification

The goal of Phase 1.0 is to swap ending nets for starting nets between blocks, so that no block need route ending and starting nets in the same column. To show that this is always possible, we outline a simple (and probably inefficient) method to achieve this swapping. Let e_i denote the number of ending nets in block i. We will make e_i a multiple of pk, by either sending $e_i \pmod{pk}$ ending nets out of the block to the right, or by receiving $pk - e_i \pmod{pk}$ additional ending nets from the right. For each ending net received by block y from block x, a non-ending net must be returned to block x from y (i.e., a starting net, vertical net, or an empty terminal). "Swapping" an empty terminal simply means that the right going ending net occupies the empty terminal in y, thereby leaving an empty terminal in x. The following procedure selects a suitable set of pairs for interblock swapping.

$total = 0$
for $i = 1$ to $|B|$ ($|B|$ represents the total number of blocks)
{ **if** $total + e_i \pmod{pk} < pk$
  { Extend $e_i \pmod{pk}$ ending nets to the right
    $total = total + e_i \pmod{pk}$
  }
  else
  { Select $pk - e_i \pmod{pk}$ right-extended ending nets
    Swap these nets with $pk - e_i \pmod{pk}$ non-ending nets from block i
    $total = total - (pk - e_i \pmod{pk})$
  }
}

Note that in this strategy, the total number of extended nets remains between 0 and pk. Also, all ending nets moved right remain ending, and non-ending nets moved left remain non-ending (vertical nets will become starting nets).

The procedure defines a set of nets to be swapped. Swaps are accomplished as follows. First, move all nets to be swapped into non-adjacent columns in layer $L-1$ (leaving layer L empty). (This can be accomplished using the Intrablock Routing Algorithm of the previous section, using at most $O(rp)$ tracks.) A single swap is performed in three tracks using layers $L-1$ and L, as shown in Fig. 6. Obviously, non-overlapping swaps can be performed using the same set of three tracks.

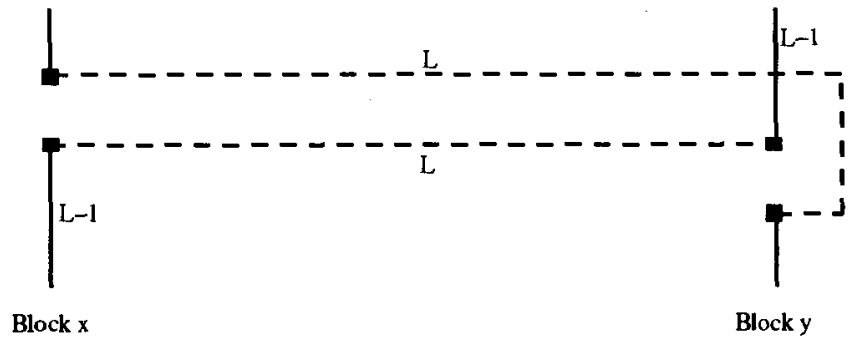


Fig. 6. Performing an interblock swap of two nets, using layers $L-1$ and L, in three tracks.

The "density" of the swapping is bounded by pk, so $3pk$ tracks suffice. Furthermore, the density of the new problem is at most $2pk$ larger than the original. This procedure is performed on both the top and the bottom sides of the channel, as illustrated in Fig. 2. The total track usage for Phase 1.0 is therefore bounded by $O(rp + pk)$.

4.1.1. Phase 1.1: approximate horizontal routing

We describe the Phase 1.1 routing algorithm for a single block B, focusing discussion on the falling nets whenever the falling and rising procedures are symmetric. In addition to describing the actual routing of nets, we also describe the updating of the pyramid from column to column. The Phase 1.1 operations are repeated for all the blocks in the channel in succession from left to right. In each block we always route all of the starting and vertical nets in the block. In addition, we route as many ending nets as possible using the remaining (leftmost) terminals of the block, and any ending net not routed is extended into the next block.

1. Exit positioned ending and extended nets.

Positioned ending and extended nets exit in the leftmost columns of block B. Within a column the falling nets are exited in order from the p highest tracks containing falling positioned ending and extended nets. At most one ending net (the exposed net) may exit from any track in each column; up to k columns may be required to exit the ending nets from a single track (recall Fig. 5). In the first column, the exiting falling net in the highest track is moved to layer 1 to route vertically, second highest to layer 2, etc., down to the pth highest track with an exiting net, which uses layer p to route vertically out of the channel. This is repeated in each successive column while positioned exiting or extended nets remain. The following procedure illustrates Step 1 for the falling nets of a single block.

```
c = 1
while ending net terminals remain
{   Let e = the number of exposed ending nets in column c
    for i = 1 to min{e, p}
    {   Select the highest exposed ending net.
        Exit the net in column c, using vertical layer i.
    }
    c = c + 1
}
```

It may be the case that there are more positioned ending and extended nets ready to exit than there are ending net terminals. In that case, we exit as many nets as are needed to occupy every ending net terminal on that shore in block B. The nets not exited in B are extended into block $B+1$. Fig. 7(a) shows an example routing for exiting nets, with $L = 5$, $p = 3$.

2. Fill vacated lanes with continuing/starting nets.

Next, we want to move lanes vacated in Step 1 into the middle of the channel. This is achieved by routing the highest falling continuing nets down into the vacated lanes. However, a track with vacated lanes cannot be refilled until positioned nets have finished exiting. None of the tracks above and including the lowest track that exited in column 1 contain a positioned ending net in column $k + 1$, because all ending nets in those tracks exit by column k . Thus in column k+1, p continuing nets that are in the highest p tracks containing falling nets can be moved into the p tracks with lanes vacated in column 1. In general, in column i ($i > k$), continuing nets from the top p tracks fill the lanes vacated by the nets that exited in column $i - k$.

In any column, at most one continuing net can be moved into or out of any given track. Thus, it may take up to k columns to fill a track's vacated lanes, and up to k columns to empty one of the p highest tracks into vacated lanes. From column $k + 1$ to $2k$, as many as pk continuing nets are routed to cover up to pk vacant lanes which correspond to the nets that exited in columns 1 to k . Once they begin, continuing nets are routed in each column in which both rising and falling nets exit. Thus, the highest p tracks that contained falling nets at the start of block B become empty by column $2k + 1$. In each successive set of k columns, the next highest p tracks are cleared by moving their continuing nets.

If more falling nets exit than rising nets, then we may need to route some starting falling nets in the same columns as falling ending nets. These starting falling nets are routed as *balancing* nets, and take the place of the continuing nets in filling in vacated lanes. The procedure for Step 2 is given below.

$c = k + 1$
while c contains exiting falling nets
{ Let e = the number of nets which exit in column c
for $i = 1$ to e
{ Select the lowest vacated lane (τ, λ) in c which contains no positioned exiting nets
if the rising net terminal in layer i contains a starting net
Route the starting (balancing) net into (τ, λ) using layer i.
else
Route highest exposed falling continuing net into (τ, λ) using layer i.
}
$c = c + 1$
}

Due to the delay of pk in refilling vacated lanes, up to pk lanes vacated by falling exiting nets are not covered by continuing nets, and become bad track lanes (see Fig. 7(b)).

3. Update the pyramid — both sides exit nets.

In columns where both sides exit nets, the pyramid must be expanded to reclaim these lanes. When both sides are exiting nets, pk new lanes become available on either side of the channel every k columns. Of course, the pyramid cannot begin reclaiming lanes exited by continuing nets immediately, but rather is delayed by up to k columns. Instead, the pyramid begins expanding into empty "good" tracks in the middle of the channel, until one full step of the continuing net staircase is cleared, at column $(2k + 1)$ in the block. Thus, the pyramid leaves up to pk lanes vacated by continuing nets unreclaimed, to become bad lanes. The pyramid expands by x paths in every column in which x rising and falling nets exit. The pyramid expansion begins in column 1 in a completely empty good track. Thus if the old pyramid had a partially filled staircase step, these unfilled paths become at most pk bad lanes. Fig. 7(c) shows the expansion of the pyramid.

4. Route starting nets.

Having completed the routing of exiting nets, the starting nets from one shore are routed directly and the starting nets from the other shore are routed through the pyramid. We choose to route the side with more remaining starting nets directly. Assume that the falling nets are routed directly, and rising nets are routed around the pyramid (the converse case is similar). Due to Phase 1.0, starting rising and falling nets begin exiting in a new segment of columns.

To route rising starting nets around the pyramid, we route the nets vertically to the lower half of the pyramid. The rightmost segment of starting nets routes into the outermost tracks of the pyramid. The nets are routed to the left in the pyramid, then across the channel, then right again in the associated track of the pyramid path on the other side of the channel. Each turn of the nets between

vertical and horizontal orientations uses the segmented organization described for the directly routed nets (see Fig. 7(d)). Thus, it is easier to describe the routing by beginning at the rightmost column segment containing starting nets (call it s), and work to the left. We match pk-lane steps of the pyramid with the steps formed by the vertical connections. If there are fewer than pk lanes in the outer group of the pyramid, then the starting nets use the first complete group in the pyramid, and so up to pk bad pyramid lanes form. In addition, since a partially filled group in the pyramid may not be able to be used immediately, the pyramid should contain an extra pk paths, using $2pk$ lanes. In the procedure outlined below, it is assumed that if necessary, the outer partial group of pyramid paths have been removed from consideration. Furthermore, the rightmost segment s is assumed to contain a full pk starting nets for simplicity.

Directly routed falling nets also form a staircase pattern, so that the rightmost segment of starting nets uses the lowest p tracks available after routing the rising nets through the pyramid. Each segment forms a step in the staircase, and the pk nets are turned from vertical to horizontal in sets of pk nets, one group above the rising nets which were routed into the pyramid. These starting falling nets fill in the pyramid lanes claimed by the starting rising nets (delayed by k columns). Thus, the first pk nets must fill in empty "good" tracks. A total of pk bad/good tracks are therefore used by the directly routed starting nets.

for $i = 1$ to s
{ Route rising net segment $s - i + 1$ into the ith lowest pyramid group, via a staircase step
Route falling net segment $s - i + 1$ into the $(i + 1)$th lowest pyramid group, via staircase step
}

5. Route vertical nets and place empty terminals.

Empty terminals do not interfere with routing of nets, but do affect pyramid updating. If there are empty terminals on one shore of block B and starting or ending nets on the other, then the pyramid must be reshaped by reassociating columns and tracks. If the other shore is entering nets, then in each segment p tracks in the pyramid (all on one side of the channel) are occupied by directly routed entering nets. The remaining tracks are reassigned to columns so that the pyramid contains only balanced, C-shaped paths. In each column, $p/2$ paths are deleted from the reshaped pyramid. If the other shore is exiting nets, the reverse process takes place: in each segment, p tracks are added to the pyramid on the exiting side; the pyramid is reshaped so that in each column, $p/2$ C-shaped paths are added to the pyramid.

The vertical nets are routed solely in the vertical routing layers, $1, \ldots, p$ from the top to the bottom shore. The routing does not interfere with the routing of starting and ending nets, and the pyramid structure is not changed.

Fig. 7(a). Phase 1.1: Exit positioned ending and extended nets (indicated by arrows). In this example, $L = 5$ and $p = 3$.

Phase 1.1 uses lanes for three purposes: To route regular non-extended nets, to route extended nets, and good/bad lanes. The reordering of pins (exiting nets before starting nets) ensures that the density of the approximate routing is not increased except by extended nets. If the number of extended nets is bounded by E

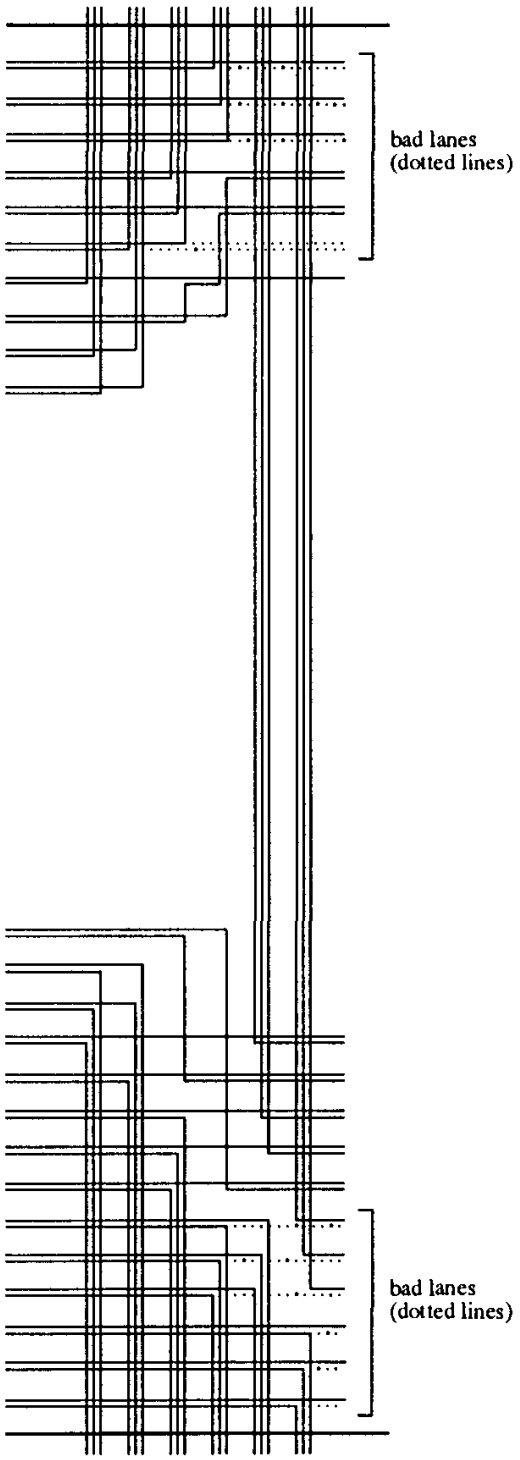


Fig. 7(b). Phase 1.1: Fill vacated lanes with continuing nets and balancing nets.

at any column, then at most $d + E$ lanes are used to route the nets. An additional set of $O(pk)$ lanes begin as good lanes in the middle of the channel and end as bad lanes distributed through the rising and falling regions. Since there are $k = L - p$ lanes per track, the total Phase 1.1 track usage is $(d + E)/(L - p) + O(p)$.

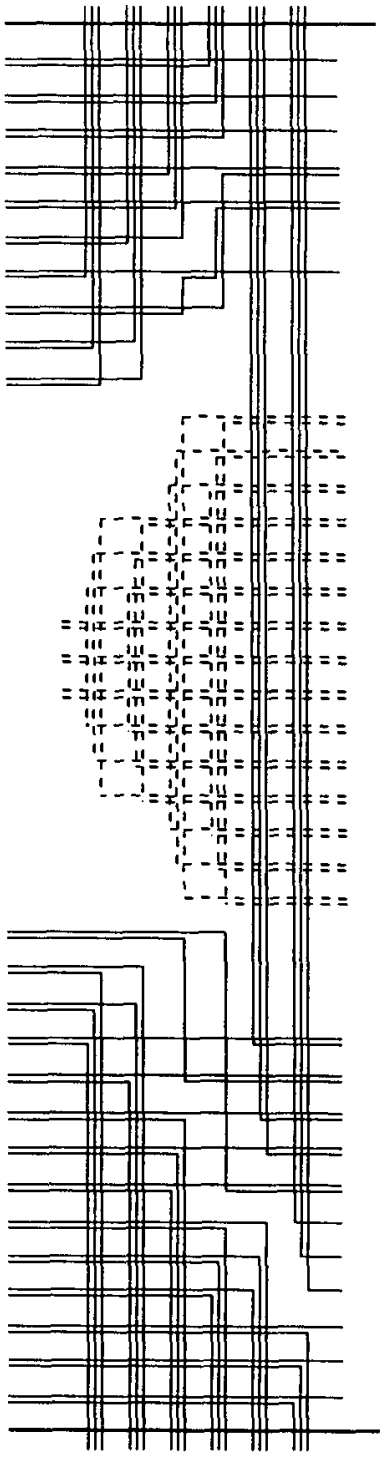


Fig. 7(c). Phase 1.1: Expand the pyramid into vacated lanes.

4.1.2. Phase 1.2: reclaiming bad tracks and reordering nets

Phase 1.2 has two purposes. The first is to move the bad (empty) lanes back into the center of the channel, where they become good lanes and are again available for

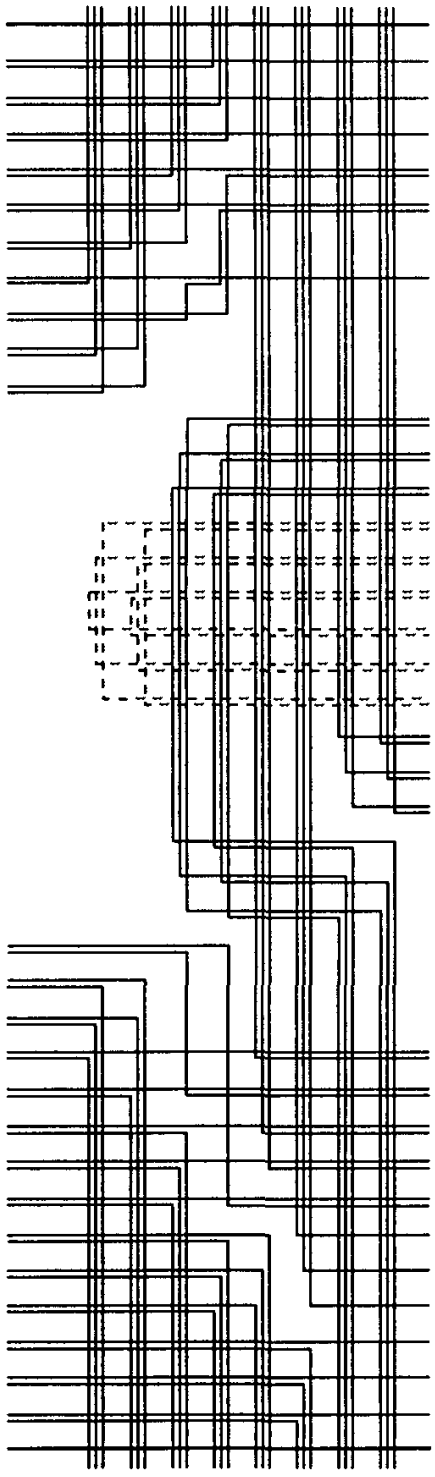
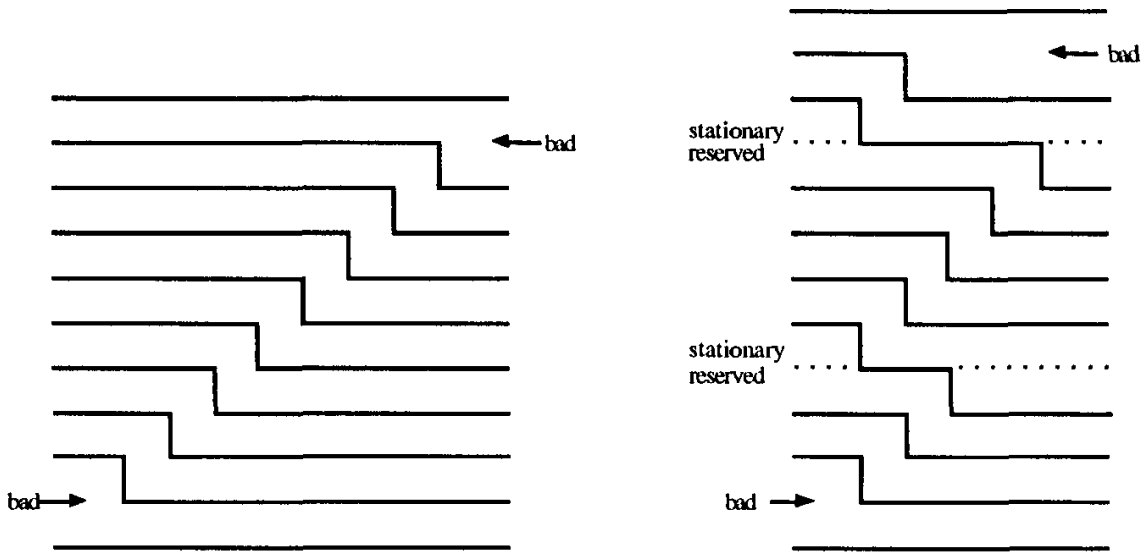


Fig. 7(d). Phase 1.1: Route starting nets, one side directly, the other through the pyramid.

horizontal routing. The second is to position nets, i.e. to reorder the nets within each column so that the nets which need to exit (ending and extended nets) are packed into the uppermost of the k horizontal layers, $p+1$ through L. Both tasks

are accomplished by vertically jogging falling (rising) nets toward the lower (upper) shore, in effect propagating empty lanes toward the middle of the channel. A similar jogging strategy is used for multilayer routing in Berger et. al.[2] The jogged lanes remain within layers $p+1$ through L, and therefore do not affect the Phase 1.1 vertical connections.



(a) Simple. (b) Using stationary reserved tracks.

Fig. 8. Propagation of a bad lane ($k = 1$).

The basic idea is illustrated in Fig. 8 for a single horizontal routing layer ($k = 1$). Notice in Fig. 8(a) that n columns are used in order to move a single bad lane n tracks up. But in order to complete the reclamation of bad lanes within a block (r columns), we must avoid using as many as d columns for each bad lane. Fortunately, the number of columns required to propagate a bad lane can be reduced by introducing extra uniformly spaced empty tracks; the spacing of these extra tracks determines the number of columns needed to complete the propagation. For instance, in Fig. 8(b) an additional empty track is introduced after every fourth track, in order to reduce to five the number of columns needed to propagate a bad lane. These additional tracks, called *stationary reserved tracks*, allow for simultaneous propagation of lanes between every pair of adjacent reserved tracks as demonstrated in Fig. 8(b).

In block B, Phase 1.2 successively moves each of the $O(pk)$ bad lanes produced in block $B-1$ into the middle of the channel, for use as good lanes in block $B+1$. (These are an addition to the good/bad lanes currently being routed by Phase 1.1.) Additionally, at each jog step we reorder the track receiving the jogged lane. To reorder a track while simultaneously moving a set of lanes, we reserve two empty tracks, called *moving reserved tracks*, and jog them along with the bad lane being reclaimed. Suppose track i contains the bad lane currently being jogged, as depicted

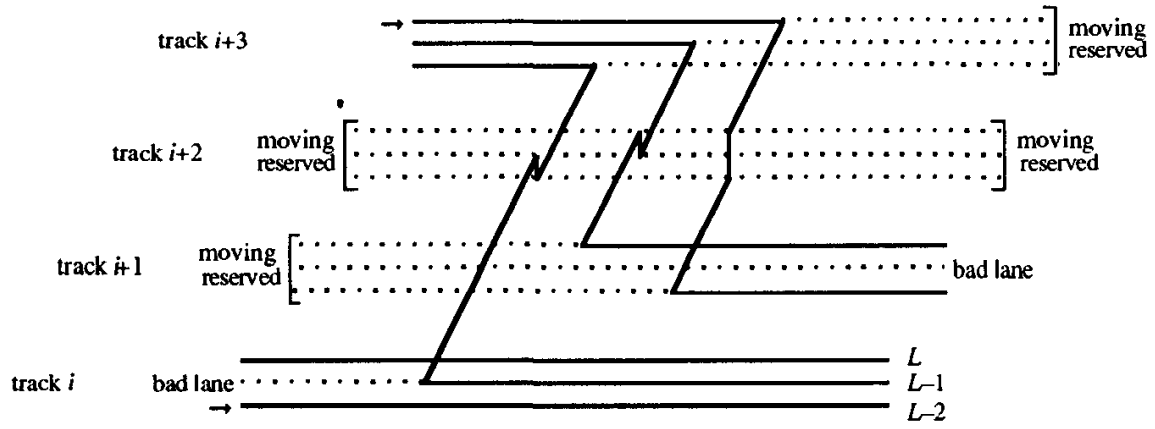


Fig. 9. Simultaneous movement of a bad lane and track reordering ($k = 3$). '→' marks falling exiting nets which need to be positioned.

in Fig. 9. Then the moving reserved tracks are tracks $i + 1$ and $i + 2$, and the bad lane (or lanes) in track i is filled in from track $i + 3$. Track $i + 2$ is used to allow the nets to change layers, and track $i + 1$ receives the overflow of nets from track $i + 3$, after track i is filled. After jogging the k nets from track $i + 3$, tracks $i + 2$ and $i + 3$ become the empty moving reserved tracks, track i is full, and track $i + 1$ now contains the bad lane. Reordering is accomplished by putting each ending and extended net into the lowest layer available in the track it moves into, and each continuing net into the highest layer available in the track it moves into. In the last bad track reclamation, at the end of the block, the reserved tracks are swept through all of the falling and rising nets so that every track can be reordered just before entering the next block. It takes up to k columns to move a bad lane (or lanes) by one track; however, a bad track which contains more than one bad lane can have all of its lanes reclaimed in the same operation.

Suppose a single bad lane (on the falling net side) is to be reclaimed. In addition to the propagation of the bad lane, an empty lane is simultaneously propagated from each stationary reserved track above track i (and below the center of the channel) to the next stationary reserved track above it. Since the bad lane(s) are simultaneously propagated, a pair of moving reserved tracks are needed for each stationary reserved track. (This allows reordering along with the simultaneous propagation.) By selecting the interval between stationary reserved tracks, we can ensure that all $O(pk)$ bad lanes can be reclaimed within the r columns of the block.

Figure 10 shows the simultaneous movement of empty tracks to reclaim a bad lane efficiently. In this example, sets of four tracks are separated by a stationary reserved track. Initially, the pairs of moving reserved track are at the bottom of each set. At the completion of the propagation, the bad lane lies in the highest falling net track. In addition, the pairs of moving reserved nets now lie at the top of the sets to which they belong. In order to ready the channel for the next bad lane reclamation, it is necessary to propagate the moving reserved nets back down to the bottom of the sets to which they belong. Of course, if the bad lanes are propagated

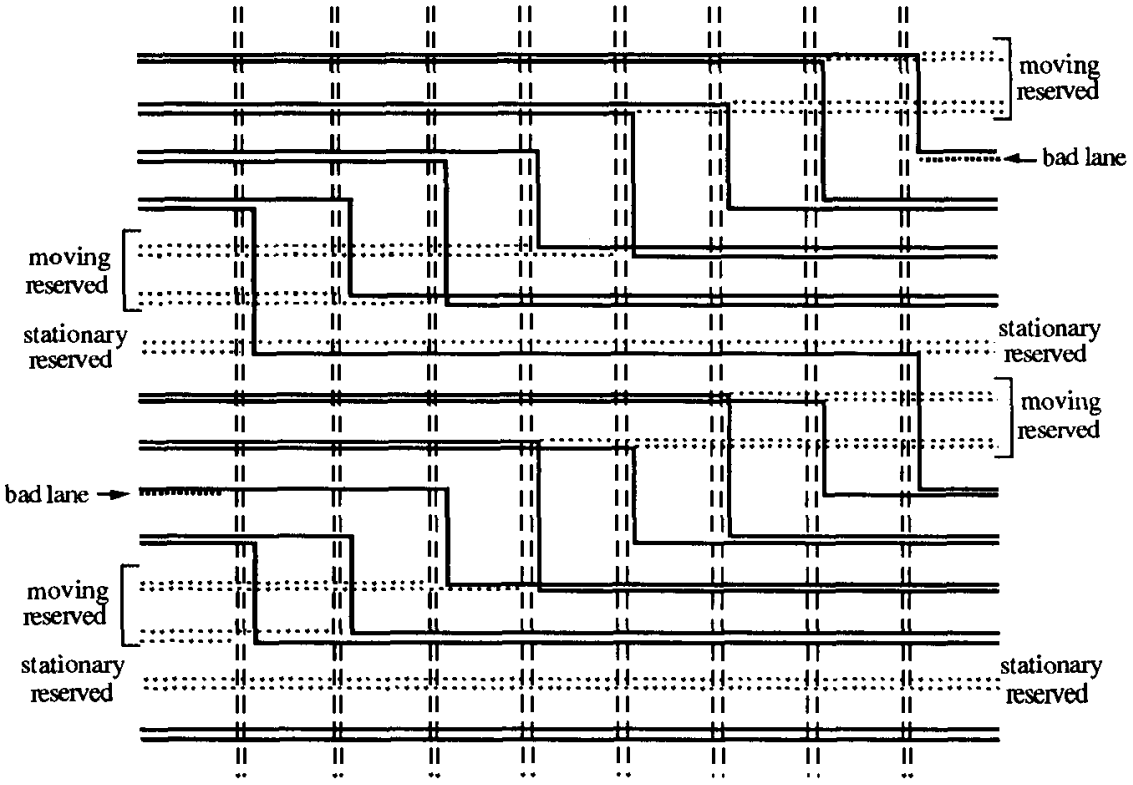


Fig. 10. Reclaiming a bad track ($L = 4$; $p = 2$). Vertical dashed lines run exclusively in layers 1 and 2, while solid horizontal wires and vertical jogs run only in layers 3 and 4.

alternately from the rising and falling sides of the channel, the falling net moving reserved tracks can be replaced while a bad lane is being propagated through the rising nets, and vice-versa. Finally, note that we can propagate the moving reserved tracks in *all* of the sets (without using any extra columns or tracks) in order to allow the reordering of *all* of the tracks in the channel.

We make a few observations about tracks which are properly ordered, that is tracks which contain no non-positioned starting or extended nets. The starting net terminals can be arranged within each block B before they enter the channel so that they are ordered within their tracks at the start of block $B+1$. A track has its order disturbed only when one of its lanes is filled with a continuing net, since bad lane reclamation maintains the proper order of tracks. Hence, at the beginning of each block, the only unordered tracks are those which have had lanes filled in with continuing nets *since their latest reordering.* We provide a bound on the number of such tracks below.

Assuming W bad tracks are to be reclaimed in each block, the reserved tracks are placed at intervals $X = r/W(k+1)$ tracks apart, with 3 reserved tracks, one stationary and two moving, placed at each X track interval. Each bad track or lane is reclaimed in a separate set of $X(k+1)$ columns: Xk columns to propagate the bad lane, and X columns to return the moving empty reserved tracks to their original location. Then $X(k+1) = r/W$ columns are used for each reordering,

and $WX(k+1) = r$ columns are used to reclaim the bad tracks for each block. In addition, the $O(pk)$ bad lanes being reclaimed into $O(p)$ good tracks are not accounted for in Phase 1.1, and must be counted in Phase 1.2. Since three reserved tracks are needed for each interval of X tracks, a total of $3/X = 3W(k+1)/r = ((k+1)/r)O(pk) = O(pk^2/r)$ reserved tracks are used per non-reserved track. Letting E denote the number of extended nets, this gives a total $(3/X)((d+E)/(L-p)+O(p))$ reserved tracks. Total track usage for Phase 1.2, summing reserved and good/bad tracks, is thus $O(pk^2/r)((d+E)/(L-p)+O(p))+O(p)$ tracks.

4.2. Track Usage for Phase 1 Algorithm

Now let us evaluate the combined performance of Phases 1.1 and 1.2. First, we evaluate the maximum number of extended nets. Let Y be the number of columns used for a reordering operation. In the last Y columns of the previous block, all tracks are reordered. Up to Yp tracks may have nets exit in those columns and subsequently become unordered. Thus, there are at most $Yp(k-1)$ non-positioned nets. The success of the reordering is measured by how well it forces nets to become positioned – this in turn is measured by the number of extended nets in the channel.

Lemma 1. *The number of extended nets in any block i never exceeds the maximum possible number of non-positioned nets within a given block.*

Proof. Let Z be the maximum number of non-positioned nets in any block of our 3dCRP, and let E_i be the number of extended nets in block i. There are no extended nets in block 1, so clearly $E_i \leq Z$. Suppose at the start of block i, there are $E_i \leq Z$ extended nets, and there are N_i ending and extended nets total in block i. The number of nets exiting in block i is lower bounded by the minimum of the number of positioned extended and ending nets, which is $(N_i - Z)$, or the number of ending terminals, $(N_i - E_i)$. Since $N_i - Z \leq N_i - E_i$, at most Z nets will not exit, so that at the start of block $i+1$ there are no more than Z extended nets. Hence, $E_{i+1} \leq Z$, which by induction holds for any block in the channel. □

This implies that the number of extended nets, E, is at most $Yp(k-1)$. The value of Y is r/W, or $O(r/pk)$, and hence $E = O(r)$. Now, we can state the following bounds on the performance of Phase 1.

Lemma 2. *For any two-terminal 3dCRP problem with p terminals per stack and L layers, $L > p$, and density $d > Lp$, the Phase 1 algorithm on a block size of r can be routed in at most*

$$d/k + O(r/k) + O(pk) + O(dpk/r) + O(rp)$$

tracks $(k = L - p)$. The density of the problem that remains to be routed by Phase 2 is $O(rp)$.

Proof. Recall that Phase 1.0 requires at most $O(pk + rp)$ tracks. We have already seen that Phase 1.1 uses $(d+E)/(L-p) + O(p)$ tracks, where E is the

maximum number of extended nets. Substituting our previous bound on E, we obtain $d/(L-p)+O(r/k)+O(p)$. For Phase 1.2, we had $O(pk^2/r)((d+E)/(L-p)+O(p))+O(p)$ tracks. Again substituting $O(r)$ for E, we obtain $O(pk^2/r)(d/(L-p)+O(r/k)+O(p))+O(p)$. The total track usage for the Phase 1 of the general 3dCRP algorithm is therefore

$$(O(pk)+O(rp))+(d/k+O(r/k)+O(p))+(O(dpk/r)+O(pk)+O(p^2k^2/r)).$$

This result simplifies because of our assumption that $d > Lp > pk$.

Next we estimate the density of the remaining subproblems after Phase 1. First consider nets which are routed to the correct block. The remaining subproblems due to these nets require only intrablock routing. There are at most rp such nets within a block, and therefore the density due to such nets is at most rp. To this we add the maximum number of extended nets that cross any given cut, which is $E = O(r)$. Thus, the remaining density is $O(rp)$. □

Finally, let us consider the special case of $p = L-1$. This is the case in which the terminals are most densely packed. However, since $L-p=1$, there is only one lane per track, and so the previous difficulties with the reordering to maintain positioned nets does not occur – nets are always positioned by definition. Therefore, there are no extended nets ($E=0$), and all nets are routed to their desired blocks. This observation yields the following corollary.

Corollary 1. *For any two-terminal 3dCRP problem with L layers, at most $p=L-1$ terminals per stack, and density $d > Lp$, the Phase 1 algorithm on a block size of r can be routed in at most $d+O(dp/r)+O(rp)$ tracks. Each net is routed to a terminal within its correct r column block.*

4.3. Track Usage for General 3dCRP Algorithm

The importance of avoiding extended nets entirely is that the remaining subproblems are contained entirely within successive r column blocks. Thus, our simple Intrablock Routing Algorithm described in Sec. 3 can be used to complete the routing. In particular, for $p = L-1$ there are no extended nets and a two phase approach will complete the 3dCRP. Accordingly, we can obtain a routing for any $p < L$ by the following strategy.

Phase 1. First apply the general Phase 1 routing procedure previously described ($p < L$), selecting a block size $r = \Theta(\sqrt{d(L-p)})$. From Lemma 2, the total track usage is therefore

$$d/k+O(\sqrt{d/k})+O(pk)+O(p\sqrt{dk}).$$

Removing the dominated terms in the order bounds, we obtain $d/(L-p)+O(p\sqrt{d(L-p)})$. As illustrated in Fig. 2, this leaves us with two subproblems, each with density bounded by $O(p\sqrt{d(L-p)})$.

Phase 2. Next, we apply the restricted $p = L-1$ algorithm to the two remaining subproblems. We can do this simply by ignoring all layers above $p+1$, that is, assume a new number of layers, $L' = p+1$. Phase 2 uses the same block size, $r = \Theta(\sqrt{d(L-p)})$, where d is the original problem density. The actual density that remains for Phase 2 is $O(p\sqrt{d(L-p)})$ (or $O(rp)$), and so the total track usage is $O(p\sqrt{d(L-p)}) + O(p^2)$. For $d > Lp$, this is dominated by $O(p\sqrt{d(L-p)})$. This leaves four subproblems (recall Fig. 2) in which each net is within its correct block of $O(\sqrt{d(L-p)})$ columns.

Phase 3. Finally, we apply the Intrablock Routing Algorithm given in Sec. 3 to complete the four block-restricted subproblems. Each subproblem uses at most $O(rp)$ tracks, and thus a total of $O(p\sqrt{d(L-p)})$ tracks suffice to complete Phase 3.

This gives the following result for 3dCRP routing.

Theorem 3. *Any two-terminal 3dCRP problem with p terminals per stack and L layers, $L > p$, and density $d > Lp$, can be routed using at most $d/(L-p) + O(p\sqrt{d(L-p)})$ tracks.*

5. Multiterminal Net 3dCRP Algorithm

In this section, we describe an extension of the two-terminal net 3dCRP algorithm to allow multiterminal nets for the general case $L > p$. The general multiterminal net 3dCRP algorithm uses a three-phase strategy as in the two-terminal net case, with small modifications to each phase to allow splitting and joining of net branches.

The main difficulty in routing multiterminal nets is that they cannot be strictly classified as either rising or falling. A common channel routing technique, which we employ, is to route portions of the net in both the rising and falling regions of the channel. Of course, the disadvantage is that twice as many lanes may be used. Consequently, the multiterminal net 3dCRP algorithm requires $2d/(L-p) + O(p(L-p)\sqrt{d})$ tracks, approximately twice the two-terminal net algorithm bound. Extended net wires may pass several terminals of their net, so there also may be multiterminal nets to route in Phase 2. Hence Phase 2 uses a *multiterminal net* $p = L-1$ version of Phase 1 to complete the interblock routing. Finally, in Phase 3, the extended multiterminal net version of the Intrablock Routing Algorithm is used; the algorithm reorders nets within each block and also merges terminals that are in the same net and in the same block.

The main extensions to the two-terminal net algorithm occur in Phase 1.1 (and hence Phase 2.1) to allow starting nets to initiate wires in both rising and falling regions, and to allow these wires to make multiple connections on the top and bottom of the channel, respectively. We present here an overview of the extensions to Phase 1.1 that are needed to route multiterminal nets. A net branch is *ending-continuing* in the current block if it has a terminal in the current block, but there are also terminals in this net in blocks further to the right on the same shore

of the channel. A net is a *starting-splitting* net if it is starting in the current block but also has terminals in subsequent blocks on both sides of the channel. A net is a *vertical-splitting net* if it has its leftmost terminals in the current block on both sides of the channel and also has terminals in subsequent blocks to the right. In Phase 1 every multiterminal net is in one the following categories in every block: starting, starting-splitting, ending, continuing, ending-continuing, extended, extended-continuing, vertical, or vertical-splitting.

The splitting operation for exiting nets is fairly simple. Ending-continuing and extended-continuing nets exit along with ending nets. The net must both exit and continue on to subsequent blocks, so the net branches into two parts. One wire branches off and exits along with other exiting ending nets; the other branch of the ending-continuing net continues in its track as if it were a continuing net. Since an ending-continuing net remains in its lane, it blocks any other nets that may be ending or ending-continuing nets in the same track. This results in a potentially larger number of non-positioned, and consequently extended, nets. However, our algorithm ensures that the number of extended nets is at most rpk.

Corresponding to the splitting of extended-continuing and ending-continuing nets when they exit is the splitting of starting-splitting and vertical-starting nets when they enter the channel. When starting-splitting and vertical-starting nets enter, they are split off in empty pyramid path tracks as they are routed vertically. The nets which are exposed to the horizontal routing layer can split off a branch into a track lane, k nets per track. When all the exposed nets have split off, we change the order of the nets in each column in $O(p)$ tracks (these tracks become bad tracks), so that there is a new set of exposed nets. The newly exposed nets are then split off, then further reordering of nets is performed, etc., until all the splitting nets in each column have branched off. After starting nets from both sides have been split off, starting nets are routed across the channel in a manner similar to the two-terminal algorithm: one side routes directly across the channel, the nets from the other side backtrack around the pyramid. Since there are p nets per column, p iterations suffice to expose and split off every net in a column, so that a total of $O(p^2)$ bad tracks are formed.

Phase 1.2 is not affected by ending-continuing nets or starting-splitting nets, so the Phase 1.2 algorithm is similar to the two-terminal net Phase 1.2 algorithm. (However, the number of bad tracks may be different, thus the number of reserved tracks required is different.) The modified algorithm outlined here results in the following bound for multiterminal net 3dCRP.

Theorem 4. *The general multiterminal net algorithm can route any 3dCRP containing multiterminal nets, with* $p < L$, *density* $d > Lp$, *in* $2d/(L-p)+O(p(L-p)\sqrt{d})$ *tracks.*

6. Conclusion

The 3dCRP problem defined here is a very general model for multilayer channel routing, and, it is hoped, should be applicable in a number of different routing

situations. The algorithms and bounds presented serve to evaluate the potential for fully exploiting multiple layers. We have shown that the simple lower bounds can be approached as long as at least one terminal per stack is unused. Furthermore, if terminals are packed only a little more densely, then non-trivial problems may require very large channel widths, dependent on the number of nets and independent of the problem density.

We have assumed here that the channels are oblong, i.e., $pL < d$. Since the number of layers is typically a small fixed value, this assumption is likely correct for many types of problems. Its strength is that it allows one to concentrate on routing efficiently in one dimension, at the expense of another. However, it would be interesting to evaluate three dimensional channel routing for the case of "squarish" channels as well. In this case, one would probably have to consider densities in two directions, across columns and across layers. This problem appears to be much more difficult than the oblong case.

For the multiterminal net algorithm, our results exhibit the doubling of channel width which is typical of even most two-layer models, while multiterminal net lower bounds are, in general, no stronger than lower bounds for two-terminal nets. Gao and Kaufmann[10] have obtained multiterminal net algorithms that route in $3d/2 + O(\sqrt{d \log d})$ tracks, in various standard channel routing models; it may be possible to apply their ideas to three-dimensional channel routing. For $p < L/2$, we should be able to improve the multiterminal net algorithm in a manner similar to that used by Berger et. al.[2] One might avoid horizontal conflicts, and therefore the pyramid, by routing rising nets up in the upper $p/2$ layers and the falling nets down in the lower $p/2$ layers.

Our best algorithms are quite complex, and we do not expect them to be used directly in practice. The best VLSI layout algorithms usually incorporate numerous heuristic strategies, tailored to the specific problem being addressed. Instead, we hope that some of our strategies will be useful in guiding the development of practical three-dimensional routing algorithms.

Acknowledgements

This research was partially supported by Office of Naval Reasearch contract N00014-88-K-0316 and Joint Services Electronics Program contract N00014-90-J-1270.

References

1. M. Brady and D. J. Brown, "Optimal multilayer channel routing with overlap", *Algorithmica* **6** (1) (1991) 83–101.
2. B. Berger, M. Brady, D. J. Brown and F. T. Leighton, "Nearly optimal algorithms and bounds for multilayer channel routing", to appear in *J. Association for Computing Machinery.*

3. S. Hambrusch, "Channel routing algorithms for overlap models", *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, **CAD-4** (Jan. 1985) 23–30.
4. K. D. Warren, J. H. Reche, W. J. Jacobi and R. M. Lea, "A 3D HDI ASP: A cost effective alternative to WSI signal processors", *Proc. IEEE Intl. Conf. on Wafer Scale Integration*, 1989, pp. 267–276.
5. R. P. W. Pease, J. C. Bravman, O. K. Kwon, S. Hong, S. C. Douglas, B. W. Langley and A. P. Paal, "High performance packaging for VLSI systems", *Proc. Stanford Conf. on Advanced Research in VLSI*, 1987, pp. 279–292 .
6. B. Baker, S. N. Bhatt and T. Leighton, "An approximation algorithm for manhattan routing", *Advances in Computing Research* **2** (*VLSI Theory*), ed. F. P. Preparata (JAI Press, Inc., Greenwich, CT, 1984) pp. 205–229.
7. S. Hambrusch, "Using overlap and minimizing contact points in channel routing", *Proc. 21st Annual Allerton Conf. on Communication, Control, and Computing*, Oct. 1983, pp. 256–257.
8. R. Shaffer, "Multilayer channel routing with stacked terminals", M. S. Thesis, University of Colorado - Boulder, May 1986.
9. D. J. Brown and R. L. Rivest, "New lower bounds for channel width", *Proc. CMU Conf. on VLSI Systems and Computations*, Oct. 1981, pp. 178–185.
10. S. Gao and M. Kaufmann, "Channel routing of multiterminal nets", *Proc. 28th IEEE Symp. on Foundations of Computer Science*, Oct. 1987, pp. 316–325. (Submitted to *JACM*.)

Algorithmic Aspects of VLSI Layout
Eds. M. Sarrafzadeh and D. T. Lee
© World Scientific Publishing Co.

ON THE MANHATTAN AND KNOCK-KNEE ROUTING MODELS

D. ZHOU*
Department of Electrical Engineering
The University of North Carolina at Charlotte
Charlotte, NC 28223

and

F. P. PREPARATA
Department of Computer Science
Brown University Providence, RI 02912

ABSTRACT

In this paper we present a detailed analysis of how grid points are used in routing two-terminal nets in a channel under the Manhattan model. Consequently, we establish a new existential lower bound on the channel width, $t \geq \sqrt{d_m^2 + 4f^2} - 6 - 2$, where t is the channel width, d_m is the density, and f is the flux of the given *channel routing problem*. Our new lower bound shows that the channel width cannot be separately bounded by either the channel density d_m or the flux f, though most practical channel routing problem can be solved by using one or two extra tracks more than that required by the channel density We also present nontrivial lower bounds on the routing area for routing multiterminal nets in *channel junctions* in both the Manhattan and knock-knee models.

Keywords: VLSI, routing, lower-bound, channel.

1. Introduction

The Manhattan and knock-knee models are the two of most widely used models in VLSI routing. The Manhattan model is the most widely used in practice due to its simplicity of implementation (Refs. 3, 4, 9, 8, 15, 5, 17). Although most heuristic routers in the Manhattan model can effectively route a *channel routing problem* (CRP) with a few tracks above that required by the channel density (Refs. 9, 6, 16, 8, 15, 5, 17), the best known theoretically provable upper bound on the channel width is $t \leq d_m + O(f)$ (Ref. 3), where t is the channel width, d_m is the density, and f is the flux of the given CRP. On the other hand, the best known lower bound is $t \geq \max\{d_m, f\}$ (Ref. 4). There exists a gap between the lower bound and the upper bound since the flux and the channel density are unrelated. In this paper we present a detailed analysis of how grid points are used in routing nets in the Manhattan model and, consequently, establish a new existential lower bound,

*This work was supported in part by NSF MIP-9110450.

$t \geq \sqrt{d_m^2 + 4f^2 - 6} - 2$. This lower bound can be further written as $t \geq d + \Theta(f^2/d)$ for $f < d$ and $t \geq d + \Omega(f)$ for $f > d$, respectively. Using the same argument for the *switch-box routing problem* (SBRP) we also establish lower bounds on the switch-box height and width.

In the knock-knee model, routing of two-terminal nets can be related to the multicommodity flow problem and optimal routings can be obtained (Ref. 12). However, to find an optimum routing for the case of multiterminal nets in the knock-knee model is an NP-complete problem. We will study the lower bound on the routing area for routing multiterminal nets in a junction routing region (Refs. 10, 11). By properly arranging the locations of terminals of nets according to the shape of the routing region and calculating the minimal capacities required at a chosen set of cross-sectional cuts, we are able to establish nontrivial lower bounds for the junction routing problems. We further compare in detail the approach for the lower bound proof in the knock-knee model with that in the Manhattan model.

This paper is organized as follows: After reviewing the background in Sec. 2, the analysis of how grid points are used in the Manhattan model is presented in Sec. 3. The new lower bounds on channel width in the Manhattan model are then established in Sec. 4. The lower bounds for multiterminal net routing in both the Manhattan and knock-knee models are presented in Sec. 5. Some open problems are proposed in Sec. 6.

2. Preliminaries

A CRP consists of two parallel rows of terminals, and a set of nets, each of which specifies a subset of terminals to be connected by means of wires. In the following only two-terminal nets are considered, unless otherwise stated. A channel of width t is a subgraph of a unit-square grid with grid points (i, j), where i is an integer and $j = 0, 1, 2, \ldots, t + 1$ (i.e., the channel is an unbounded horizontal strip). The horizontal lines are called *tracks* and the vertical lines are called *columns*. Track 0 and track $t + 1$ are called the *lower shore* and the *upper shore*, respectively. All terminals of nets are located at grid points on either the lower shore or the upper shore and nets are routed in the edges of grids. In the Manhattan model two nets are allowed only to cross at a grid point, and in the knock-knee model, two nets are allowed both to jog (also called the knock-knee) and to cross. In both models edge-overlap is not allowed (Fig. 1).

We shall consider a *channel window* of *span* s, i.e., the portion of the channel defined by columns $1, 2, \ldots, s$. Column 1 is called the *left side* of the channel window, and column s the *right side*. A net is called a *crossing* net if it has one of its two terminals to the left of the left side of the channel window and the other one to the right of the right side. A net with its two terminals in different columns is conventionally called a *nontrivial* net and is a *trivial* net otherwise. A column is *nontrivial* if it contains at least one terminal of nontrivial nets and it is *trivial* otherwise. We use $d(i)$ to denote the density of column i, which is the number of

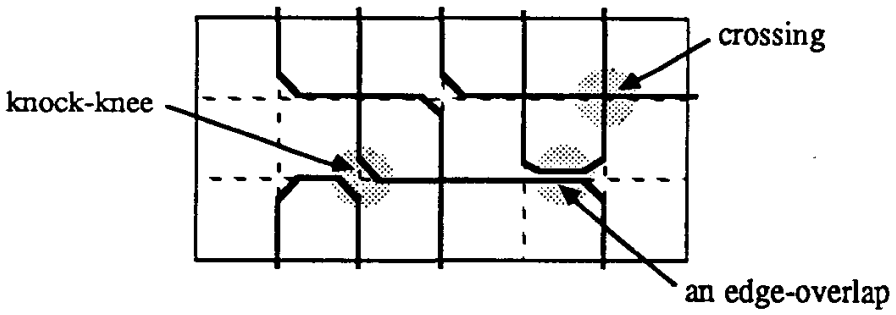


Fig. 1. The Manhattan and knock-knee models.

nets with one terminal to the left of column i and another terminal to the right of column i. The density, d_m, of a CRP is defined as

Definition 1. $d_m = \max\{d(i) : \text{integer } i\}$.

Obviously, the density d_m of a CRP is a trivial lower bound on the channel width since wires are not allowed to overlap.[a] Another lower bound, which is established by Refs. 3, 4, is the so-called flux. The flux f of a CRP is defined as follows:

Definition 2. A CRP has flux f if f is the largest integer such that there is a horizontal cut of the channel which spans $2f^2$ nontrivial columns and splits (cuts) at least $2f^2 - f$ nontrivial nets (Ref. 3).

For example, the horizontal cut shown in Fig. 2 splits 9 nontrivial nets and spans 11 nontrivial columns. Inspection reveals that this problem has flux 3. The vertical cut c shown in Fig. 2 splits 5 nets. Inspection reveals that this problem has

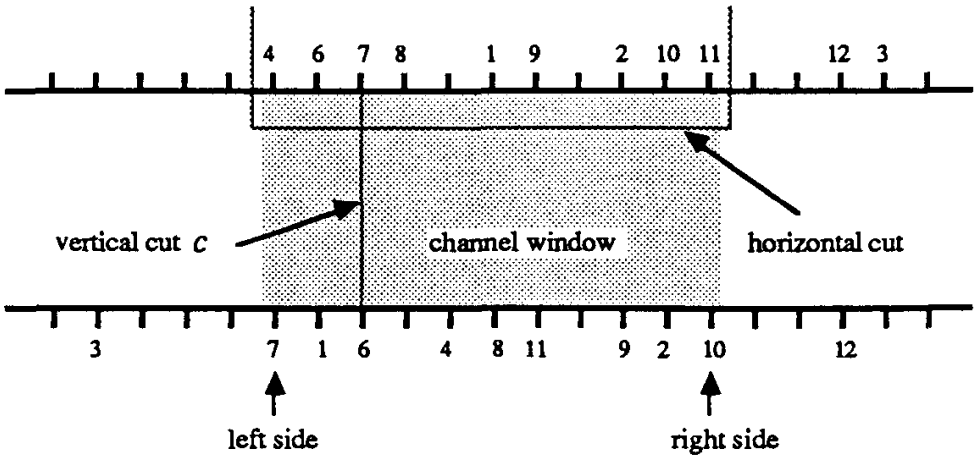


Fig. 2. A horizontal cut and a vertical cut.

[a] In the Manhattan model, $d_m + 1$ is a trivial lower bound since two nets are not allowed to bend at the same grid point.

the maximal density 5. Net 3 is a crossing net; net 12 is a trivial net; net 1 is a nontrivial net. The indicated channel window has span $s = 11$. In this paper we shall consider only nontrivial nets having one terminal on the lower shore and the other on the upper shore in different columns of the channel window. A nontrivial net is always thought of as being directed from its lower terminal to its upper terminal. Since our analysis considers a channel window, we will use only the portion of a crossing net which is inside the window. We, therefore, modify the definition of a crossing net as follows: A crossing net is one which has one of its two terminals on the left side and another on the right side of the window. The terminals of a crossing net are not assigned to any specific track. A crossing net will not be treated as a nontrivial net in this paper. Since only nontrivial and crossing nets are used in our argument, no other type of net is assumed to be present in the channel. The channel grid points are classified as follows:

Definition 3. A grid point $p = (i, j)$ is *occupied* (and *unoccupied* otherwise) if:

1. There is a vertical edge, incident to p, of a nontrivial net N entirely routed inside the channel;
2. No other vertical edges of N are incident to grid points on the same track to the left of p.

An unoccupied grid point is also called a *hole*. In other words, an occupied grid point is the leftmost one among those where a net is incident on a track.

The above definition is illustrated in Fig. 3 where the shaded region depicts the channel window. In the figure, ν_{nt} and ν_{cr} are the numbers of nontrivial and

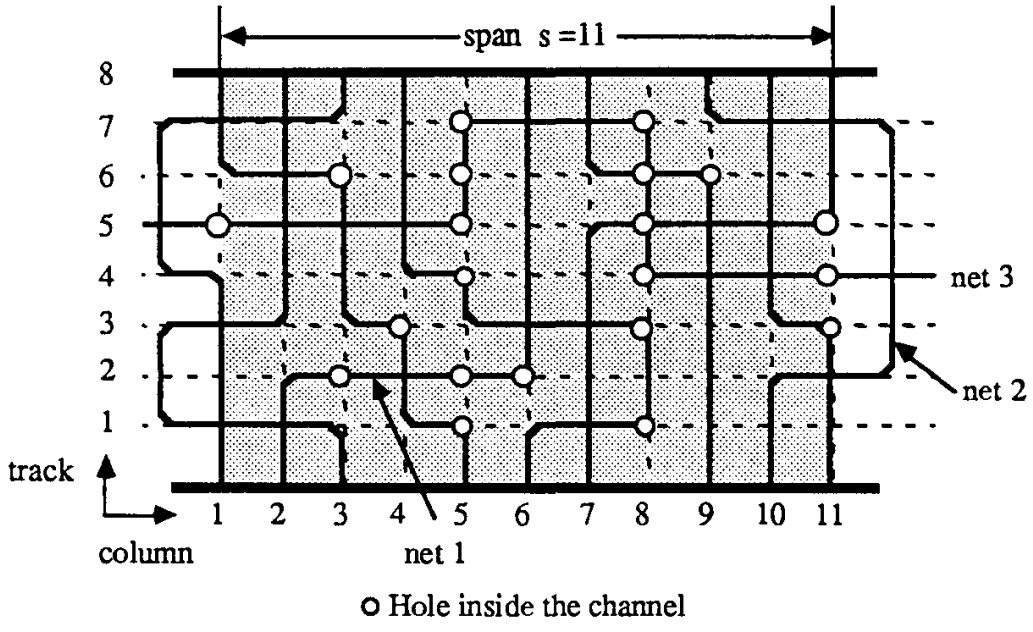


Fig. 3. A channel window with span $s = 11$, $t = 7$, $\nu_{cr} = 1$, $\nu_{nt} = 9$, and $d_m + 1 = d(3) + 1 = 5$.

crossing nets, respectively. $d(3)$ is the density at column 3. Net 1 (a nontrivial net) has its vertical edges incident on track 2 at columns 2 and 6, but only the left one, $(2, 2)$, is shown as an occupied grid point. Since occupied grid points occur only on nontrivial nets, none of the grid points to which the vertical edges of the crossing net 3 are incident (such as $(5, 5)$) are classified as occupied.

Figure 4(a) illustrates the occupied grid points contributed by nontrivial nets routed entirely inside the channel window. However, a nontrivial net can be partially routed outside the channel window and can leave and reenter the window several times as shown in Fig. 4(b). If a net N leaves the window at track j_1 and reenters it for the last time at track j_2, with $j_2 \geq j_1 + 2$, we say that the vertical interval $[j_1 + 1, j_2 - 1]$ is an *interdiction interval* for N. We then modify the definition of the occupied grid point as follows:

Definition 4. The nontrivial net N, partially routed outside the channel window, does not contribute any occupied grid point on the tracks of its interdiction intervals.

This concept is illustrated in Fig. 4(b).

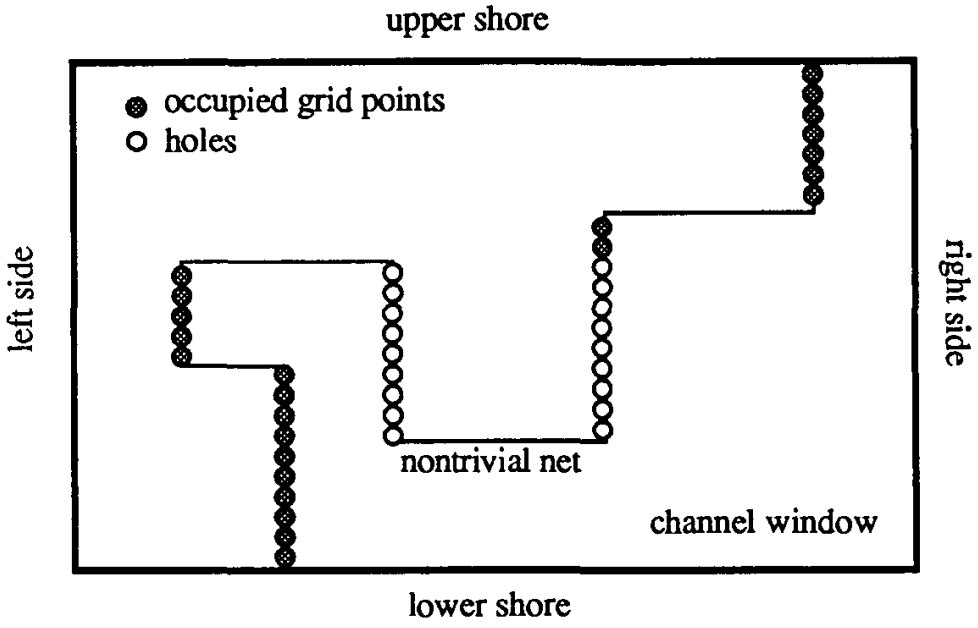
Definitions 3 and 4 give an intuitive meaning to the notion of a hole. A hole is used by a jog of a nontrivial net or by crossing nets. Whereas a hole cannot be shared by a jogging nontrivial net and any other net, it can be shared by two crossing nets. In Fig. 3, hole $(6, 2)$ is used by net 1 (a nontrivial net), and hole $(8, 5)$ is used by net 3 (a crossing net).

The lower bound sought on t is based on the idea of using holes as a crucial commodity to accomplish channel routing. As we have seen, holes are needed for the unavoidable jogs of nontrivial nets as well as for the routing of crossing nets. The supply of holes, on the other hand, depends upon the size of the channel window and the number of nontrivial nets; moreover, the hole supply can be increased by decreasing the number of occupied grid points, i.e., by routing some nontrivial nets outside the window. This strategy, however, exhibits an obvious tradeoff, since the partial external routing of nontrivial nets poses a demand on the minimum length of the channel window sides, i.e., on the minimum value of t.[b] Therefore, the balance of supply and demand of holes, as well as the tradeoff between the hole-supplying mechanism and the widening of the channel, provide the lower bound argument.

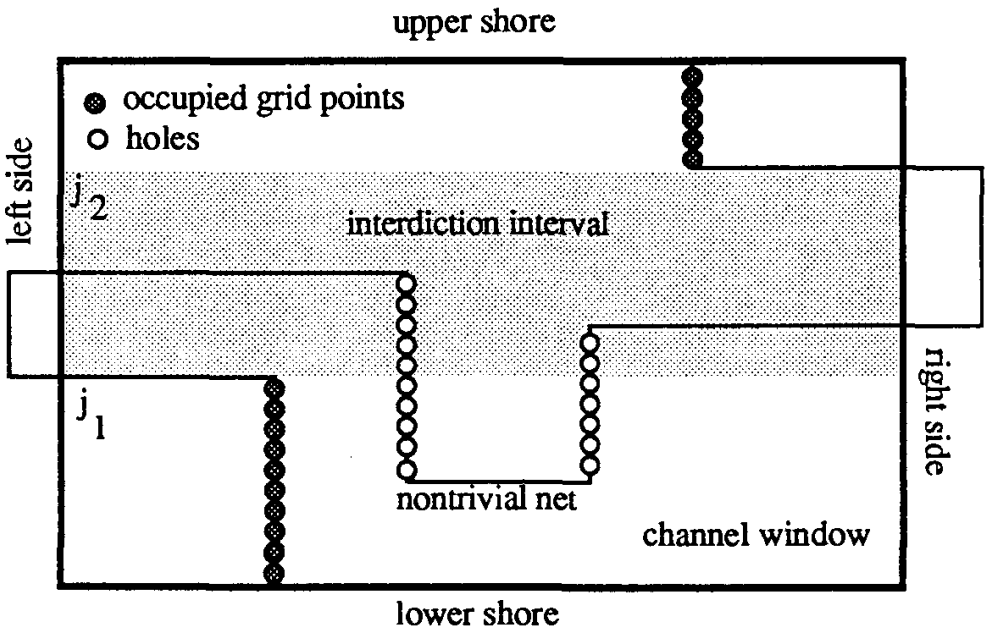
3. Analysis on the Use of Grid Points

Let us now consider a channel window of span s. Suppose there are ν_{nt} nontrivial nets, ν_{cr} crossing nets, and t tracks in this channel (Fig. 3). Necessarily, the number of holes occurring in the channel should be no less than the number of holes required to solve the routing problem. We first count the number of holes which could occur in the channel, and then count the number of holes needed for routing. Let h_i be

[b] In the following we will refer a channel window by a window if the context is clear.



(a) Occupied grid points contributed by a nontrivial net totally routed inside the channel.



(b) Occupied grid points contributed by a nontrivial net partially routed outside the channel.

Fig. 4. The occupied grid points contributed by a nontrivial net.

the number of holes in track i. The following facts are important in our counting argument:

1. $h_1 = s - \nu_{nt}$, since only a nontrivial net can contribute an occupied grid point in track 1, and each nontrivial net contributes exactly one occupied grid point.
2. Each $h_i, i = 1, 2, \ldots, t$, is at least as large as h_1.
3. $h_i = h_{i-1} + x$, where x is the difference between the numbers of nontrivial nets leaving the channel window at track $i - 1$ and those reentering the window at track i.

Each nontrivial net having left the window will reenter it, as required by the routing problem. Among such nets partially routed outside the window, we define "reentrant" nets as follows:

Definition 5. A *reentrant net* is a nontrivial net that is partially routed outside the window, and has a jog at track i with which it leaves the window and another one at track j $(j > i)$ with which it reenters the window.

Let $2\nu_{re}$ be the number of reentrant nets.[c] We prove the following lemma.

Lemma 1. *For a given CRP with span s, width t, and ν_{nt} nontrivial nets, at most $t(s - \nu_{nt} + \nu_{re}) - \nu_{re} - \lfloor \frac{\nu_{re}^2+1}{2} \rfloor$ holes can occur inside the channel window.*

Proof. The total number of holes inside the channel window will be $t(s - \nu_{nt})$ if there are no nontrivial nets crossing the sides of the window (i.e., partially routed outside the window). The number of holes, however, can be increased by routing some nontrivial nets partially outside the window as reentrant nets. If a reentrant net leaves the window at track i and reenters it at track j $(j > i)$, $j - i - 1$ holes will be created since this net will not cross tracks $i+1, \ldots, j-1$ inside the window. Obviously, the following assignment policy of exit/reentry tracks for reentrant nets maximizes the number of created holes: nets are successively assigned, and the exit/reentry tracks of the current net are chosen as the farthest ones available on the same side of the channel window (Fig. 5). It is almost trivial to show that this policy yields the following number of created holes (Ref. 4):

$$\nu_{re}t - \left\lfloor \frac{\nu_{re}^2 + 1}{2} \right\rfloor - \nu_{re} .$$

Adding this number to $t(s - \nu_{nt})$ establishes the upper bound on the number of holes that may occur inside the channel window. □

Lemma 1 establishes an upper bound on the supply of holes in a channel window. The following lemmas establish lower bounds on the demands on holes posed by nontrivial and crossing nets.

[c]For simplicity we assume that there are even number of reentrant nets.

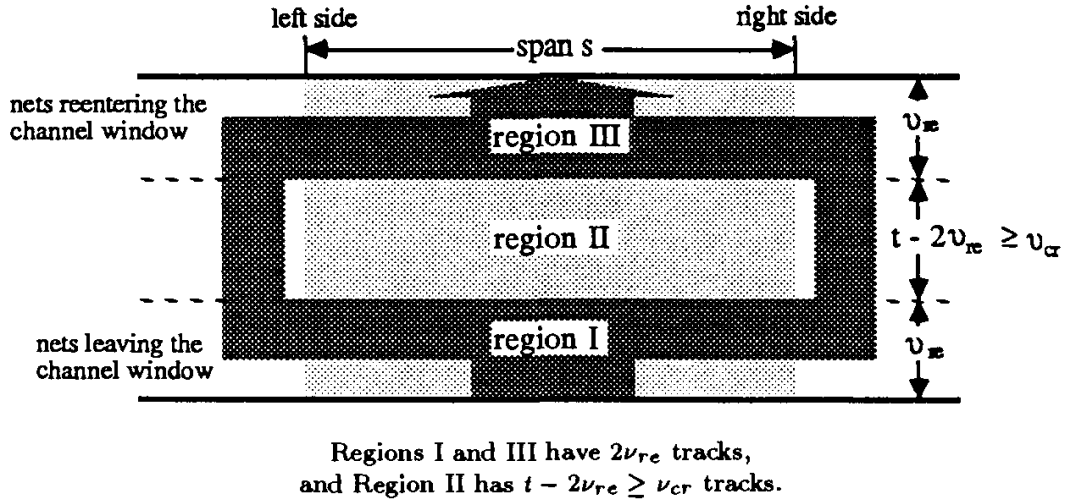


Fig. 5. A policy yielding the maximum number of holes in a channel by routing nontrivial nets partially outside the channel.

Lemma 2. *A nontrivial non-reentrant net uses at least one hole.*

Proof. Since routing a nontrivial net entirely inside the window requires at least two jogs on the same track, by the definition of holes and their use, the rightmost one of these two jogs must occur at a hole. □

Theorem 1. *At most $2\nu_{re}$ nontrivial nets can be routed without using any hole.*

Proof. A reentrant nontrivial net N has no more than two jogs inside the window. The vertical edge of N incident to one such jog is unique on the given track and, therefore, by Definition 3, does not correspond to a hole. Since there are at most $2\nu_{re}$ reentrant nets, the theorem is established. □

As illustrated in Fig. 3, net 2 (a nontrivial net) leaves the channel on the right side at track 2 and reenters at track 7. Grid points (10, 2) and (9, 7), according to Definition 3, are occupied. Then, net 2 is routed without using any holes.

In order to count the number of holes used by a crossing net, we establish the following lemma.

Lemma 3. *A crossing net uses at least $\lceil \frac{h_1}{2} \rceil$ holes, where h_1 is the number of holes in the first track.*

Proof. Consider the path of a crossing net N, directed from its entry point on the left side of the window to its exit point in the right side. Each grid point on this path is classified either as occupied or as a hole. Some of these holes correspond to intersections with other crossing nets (indeed the intersection of two crossing nets is always a hole). In this case we say that N uses "half" of that hole (partially

used hole), whereas in all other cases N fully uses a hole on its path. We wish to establish a lower bound on the use of holes by N. This lower bound is obtained by subtracting from the number of grid points on the path of N in the window an upper bound on the number of occupied grid points and one half of an upper bound to the number of partially used grid points.

Let l be the length of N. Let l_g ($l_g = l+1$) denote the number of grid points of the path of N internal to the channel; also let l_1 denote the minimum length of any path having the same extreme points as the path of N. Obviously, $l \geq l_1 \geq s-1$. Consider now a nontrivial net N' intersecting N k times ($k > 0$), and let the intersection grid points be $p_1, p_2, \ldots, p_k$, respectively. If N' is routed entirely inside the channel, then k is odd. Otherwise, k may be even. When k is odd, p_k is conservatively considered as occupied. We pair the remaining points as (p_1, p_2), (p_3, p_4), ... We now analyze a generic pair (p_j, p_{j+1}) and distinguish between the following cases:

(c1) Both p_j and p_{j+1} occur in vertical segments of the path of N. In this case, by Definition 3, both p_j and p_{j+1} are holes because the vertical edges incident to them do not belong to nontrivial nets.

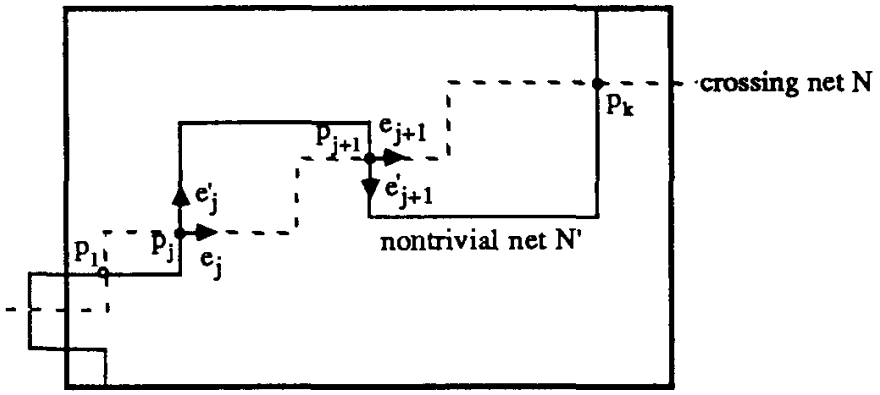
(c2) p_j occurs in a vertical segment and p_{j+1} in a horizontal segment of the path of N (or *vice-versa*). In this case, again p_j is a hole (by the above argument).

(c3) Both p_j and p_{j+1} occur in horizontal segments of the path of N. Let e_i be the edge of N incident to p_i ($i = j, j+1$), and let e'_i be the homologous edge of N'. It is easy to recognize that if e_j and e_{j+1} have identical directions, then e'_j and e'_{j+1} have opposite directions (and *vice versa*). Therefore, we distinguish between two subcases:

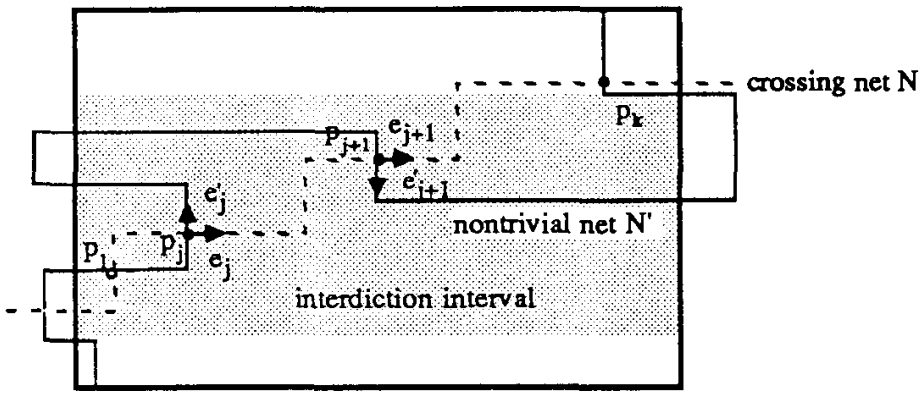
  (c3.1) e_j and e_{j+1} have identical directions. Then, a straightforward case analysis shows that for at least one of grid points p_j and p_{j+1}, either there is another grid point on the same track occupied by N' (Fig. 6(a)) or the track belongs to an interdiction interval of N' 6(b)). In either case, there is a hole at that grid point, by Definitions 3 and 4.

  (c3.2) e_j and e_{j+1} have opposite directions, and suppose that e_{j+1} is directed right-to-left (the other case is analogous). Then the column containing p_{j+1} intersects N at least three times, i.e., l is at least two units larger than l_1 (Fig. 6(c)), and this holds for each such pair although both p_j and p_{j+1} may be occupied grid points.

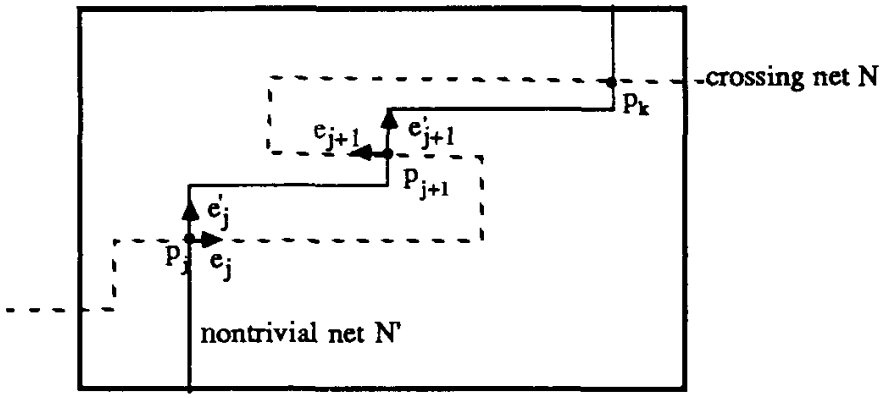
Let π_2, $\pi_{3.1}$, and $\pi_{3.2}$ be the numbers of pairs corresponding to Cases (c2), (c3.1), and (c3.2), respectively, generated by all nontrivial nets intersecting N, and let π_0 be the number of nontrivial nets having an odd number of intersections with N. Since the number of grid points that N shares with nontrivial nets is at least $\pi_0 + 2(\pi_2 + \pi_{3.1} + \pi_{3.2})$, there are at most $l_g - \pi_0 - 2(\pi_2 + \pi_{3.1} + \pi_{3.2})$ grid points that N may share with other crossing nets. At most, half of these, i.e., $(l_g - \pi_0 - 2(\pi_2 + \pi_{3.1} + \pi_{3.2}))/2$ can be used by other crossing nets. Moreover, there are at most $\pi_0 + \pi_2 + \pi_{3.1} + 2\pi_{3.2}$ occupied grid points on the path of N. In



(a) Case (3.1) There is another grid point on the same track occupied by N'.



(b) Case (3.1) The track belongs to an interdiction interval of N'.



(c) Case (3.2) l is at least two units larger than l_1.

∘ — hole
• — occupied gid point

Fig. 6. A crossing net uses at least $\lceil \frac{h_1}{2} \rceil$ holes.

conclusion, the number h of holes used by N is bounded from below as follows:

$$h \geq l_g - \frac{l_g - \pi_0 - 2(\pi_2 + \pi_{3.1} + \pi_{3.2})}{2} - (\pi_0 + \pi_2 + \pi_{3.1} + 2\pi_{3.2}) = \frac{l_g}{2} - \frac{\pi_0}{2} - \pi_{3.2} \ . \quad (1)$$

However, from the discussion of Case (3.2) we have $l \geq l_1 + 2\pi_{3.2}$, so that

$$h \geq \frac{l_1 + 1}{2} - \frac{\pi_0}{2} \geq \frac{s - \nu_{nt}}{2} \geq \frac{h_1}{2} \quad (2)$$

since $l_1 \geq s - 1$, $\pi_0 \leq \nu_{nt}$, and $s - \nu_{nt} = h_1$, by definition. □

4. Lower Bound on the Channel Width

In this subsection we discuss the lower bound on the channel width using the result of the previous subsection. We first present a lower bound based on the numbers of holes, crossing nets and nontrivial nets. This is a universal lower bound which can be applied to any CRP. However, this lower bound, like the lower bound based on flux f, is not a trivial lower bound. (Recall that the density d_m provides a trivial lower bound.) We then extend the obtained lower bound to a class of problems to establish an existential lower bound using the parameters of the density and flux. Finally, we extend our result to the case of switch-box routing.

We state the lower bound on the channel width in the Manhattan model by means of the following theorem.

Theorem 2. *The lower bound on the channel width for the two-terminal net CRP in the Manhattan model is*

$$t \geq \sqrt{(\nu_{cr} + h_1)^2 + 2\left(\nu_{nt} - \left\lfloor \frac{h_1}{2} \right\rfloor\right)} - h_1 \ , \quad (3)$$

where h_1, ν_{cr} and ν_{nt} are respectively the numbers of holes in the first track, crossing nets and nontrivial nets in the channel window.

Proof. By Theorem 1 and Lemmas 2 and 3, the total number of holes required in the channel window for completing a CRP is at least $\nu_{cr}\lceil \frac{h_1}{2} \rceil + \nu_{nt} - 2\nu_{re}$. Because, as shown in Lemma 1, at most $t(h_1 + 2\nu_{re}) - 2\nu_{re}^2 - 2\nu_{re}$ holes can occur in the channel, the following relation must be satisfied,

$$t(h_1 + 2\nu_{re}) - 2\nu_{re}^2 - 2\nu_{re} \geq \nu_{cr}\left\lceil \frac{h_1}{2} \right\rceil + \nu_{nt} - 2\nu_{re} \ , \quad (4)$$

i.e.,

$$t \geq \frac{\nu_{cr}\lceil \frac{h_1}{2} \rceil + \nu_{nt} + 2\nu_{re}^2}{h_1 + 2\nu_{re}} \ . \quad (5)$$

The right side of Formula (5) is plotted in Fig. 7 as a function Q_1 of ν_{re}. This is a relation derived from the argument of the supply and the demand of holes. We mentioned earlier that the nets which pass the sides of the channel also set a lower bound on the channel width. This requirement, referring to Fig. 5, can be formulated as

$$t \geq \nu_{cr} + 2\nu_{re} \,. \tag{6}$$

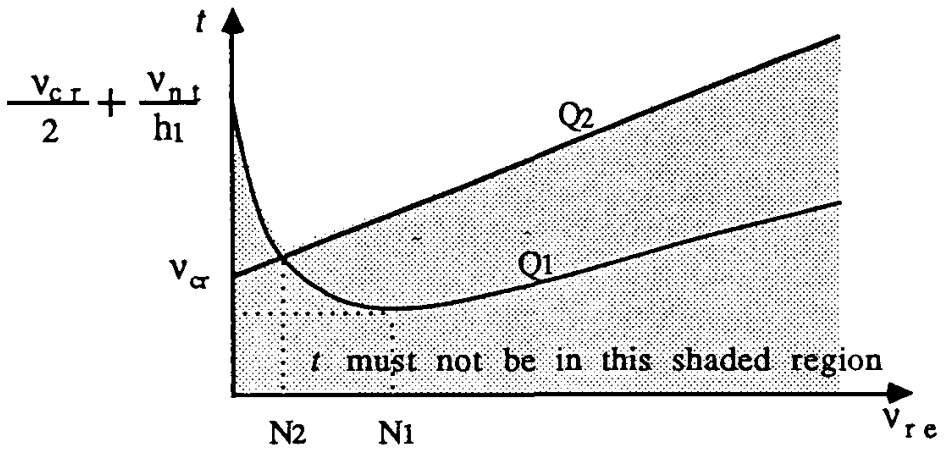
The right side of Formula (6) is also drawn as Q_2 in Fig. 7 with respect to ν_{re}. It can be seen that the lower bound achievable from relations (5) and (6) is obtained when ν_{re} takes the value at which two curves intersect in Fig. 7. This value of ν_{re} is

$$\nu_{re} = \frac{\sqrt{(\nu_{cr} + h_1)^2 + 2(\nu_{nt} - \lfloor \frac{h_1}{2} \rfloor)} - (\nu_{cr} + h_1)}{2} \,. \tag{7}$$

Substituting Formulas (7) into (6) we obtain

$$t \geq \sqrt{(\nu_{cr} + h_1)^2 + 2\left(\nu_{nt} - \left\lfloor \frac{h_1}{2} \right\rfloor\right)} - h_1 \,. \tag{8}$$

□



Q_1 is the value of the right side of Formula (5) as a function of ν_{re}.
Q_2 is the value of the right side of Formula (6) as a function of ν_{re}.
N_1 is the value of ν_{re} which sets the right side of Formula (5) to be minimum.
N_2 is the value of ν_{re} for a lower bound.

Fig. 7. Determine the lower bounds by using Formulas (3.5) and (3.6).

We can derive the known lower bound based on the channel density and flux from Theorem 2.

Corollary 1. *The lower bound on the channel width in the Manhattan model is* $t \geq \max\{d_m, f\}$.

Proof. Let the channel window consist of one column where the column density is the maximum over all column densities in the channel. Hence, $\nu_{nt} = 0$, $\nu_{cr} = d_m$, and $h_1 = 0$. From Theorem 1 we obtain $t \geq d_m$. Let the channel window contain f^2 columns which contain at least $2f^2 - h_1$ ($h_1 \leq f$) nontrivial nets. Hence, $\nu_{nt} = 2f^2 - h_1$, $h_1 \leq f$ and $\nu_{cr} \geq 0$. By plotting the right side of Formula (3) as a function of h_1 in Fig. 8 we see that $t \geq f$ for $0 \leq h_1 \leq f$. □

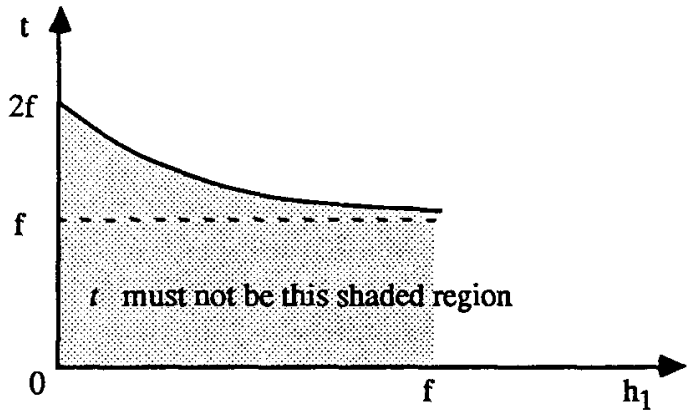


Fig. 8. Deriving the known lower bounds.

Using Theorem 2 we show the following existential lower bound on the channel width.

Theorem 3. *The existential lower bound on the channel width for the two-terminal net CRP in the Manhattan model is*

$$t \geq \sqrt{{d_m}^2 + 4f^2 - 6} - 2 ,$$

where d_m *is the channel density, and* f *is the flux.*

Proof. In Formula (3) we choose $s = 2f^2$, $\nu_{cr} = d_m - 2$, and $\nu_{nt} = 2f^2 - 2 (f \geq 1)$, as shown in Fig. 9, and obtain

$$t \geq \sqrt{d_m^2 + 4f^2 - 6} - 2 . \tag{9}$$

To complete the proof we show in the above constructed routing problem that d_m and f are two independent variables. First, we show that f is independent of d_m. In the constructed example there are $s = 2f^2$ columns and $2f^2 - 2$ nontrivial nets in the channel window, as well as $\nu_{cr} = d_m - 2$ nets crossing the window. We have to show that the flux is independent of the number of crossing nets which is two units

less than the density (Fig. 9). Suppose that the problem has flux f', which is larger than f due to adding of crossing nets to increase the density. According to the definition of the flux, there must exist a horizontal cut which spans $2f'^2$ columns and splits at least $2f'^2 - f'$ nontrivial nets. Let Z be such a horizontal cut. Note that any horizontal cut which cuts $s + x$ (x is an integer) nontrivial nets spans at least $s + 2x$ columns from the construction of the problem (Fig. 9). Therefore, Z spans $2f'^2 = s+2x$ columns, and should cut at least $2f'^2 - f' = 2f^2 + 2x - \sqrt{f^2 + x}$ nontrivial nets. Since $2f'^2 - f' = 2f^2 + 2x - \sqrt{f^2 + x} < 2f^2 - f$ for any integer x we see that there is no flux which is larger than f. Next, we show that d_m is independent of f, i.e., no matter how many nontrivial nets may be inserted into the channel window to increase the flux the density is not affected, as is trivially seen from the construction of the example. Theorem 3 is proved. □

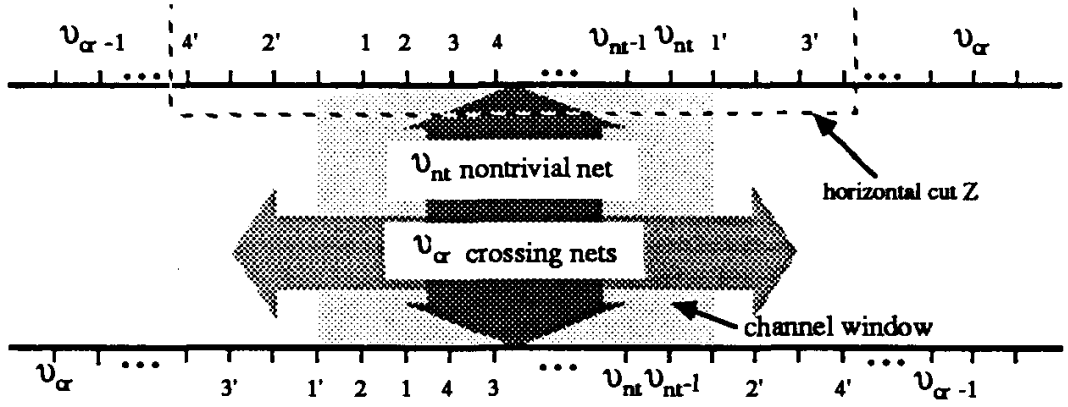


Fig. 9. Deriving the new lower bound.

The argument on the supply and the demand of holes in a chosen region to route the specified nets can be extended to the switch-box routing problem (SBRP). For the CRP we have shown the lower bound on the channel width which is measured in the y-direction. We add a superscript y to the notations introduced above for the CRP to indicate that those notions are defined with respect to the y-direction of a switch-box. Thus, t^y is the width in the y-direction of the switch-box; ν_{cr}^y is the number of crossing nets which have one terminal on the left side and another terminal on the right side; ν_{nt}^y is the number of nontrivial nets which have their terminals, one on the upper shore and another on the lower shore, in different columns; h_1^y is the number of holes in the first track; and d_m^y is the maximum density over all columns. Similarly, we define parameters, t^x, ν_{cr}^x, ν_{nt}^x, h_1^x, and d_m^x, with respect to the x-direction. In SBRPs, all terminals on four sides are fixed and no nets are allowed to be routed outside the switch-box. We construct an SBRP as shown in Fig. 10 with terminals on four sides fixed, where $\nu_{cr}^x = \nu_{nt}^y = d_m^x - 2$, $\nu_{cr}^y = \nu_{nt}^x = d_m^y - 2$. Thus, $h_1^y = t^x - (d_m^x - 2)$ and $h_1^x = t^y - (d_m^y - 2)$. Choosing the windows W^y and W^x (Fig. 10) and arguing similarly as to the case of CRPs, we obtain

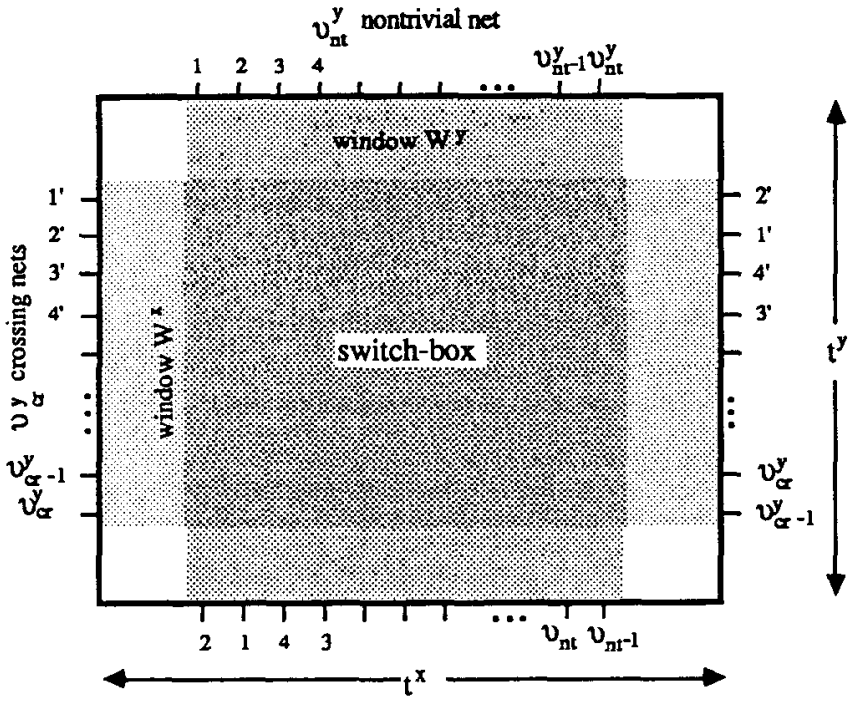


Fig. 10. The lower bounds on the width and height of switch-box.

Theorem 4. *For an SBRP, existential lower bounds on the widths in the x- and y-directions of the switch-box are*

$$t^y \geq \sqrt{(d_m^y)^2 + 4(f^y)^2 - 6} - 2 \tag{10}$$

and

$$t^x \geq \sqrt{(d_m^x)^2 + 4(f^x)^2 - 6} - 2 \tag{11}$$

A lower bound on the routing area of SBRP can be established by using the above theorem. Let $\mathcal{A}$ denote the routing area. We obtain

$$\begin{aligned} \mathcal{A} \geq t^x t^y \geq & \sqrt{\{(d_m^y)^2 + 4(f^y)^2 - 6\}\{(d_m^x)^2 + 4(f^x)^2 - 6\})} \\ & - o(\sqrt{\{(d_m^y)^2 + 4(f^y)^2 - 6\}\{(d_m^x)^2 + 4(f^x)^2 - 6\}}) \end{aligned} \tag{12}$$

5. Lower Bounds for Routing Multiterminal Nets

So far in this paper we have focused on the two-terminal net routing problems in the Manhattan model. The results obtained are based on the argument of the supply and the demand of the unoccupied grid points in a chosen routing region.

This argument cannot be applied to the knock-knee model since two nets are allowed to jog at the same grid point in this model. In fact, the channel density is both a lower and an upper bound on the channel width for routing two-terminal nets in the knock-knee model. In the following, we briefly discuss how to use the properties of multiterminal nets and the shape of routing regions to establish lower bounds on the routing area. This argument can be applied to both the Manhattan and knock-knee models.

Let us first examine a simple routing problem as shown in Fig. 11, where the routing region is a rectangle with terminals of nets in its three sides. All nets are 3-terminal nets. We check the density at a set of cross-sectional cuts $C = (c_1, c_2)$ (Fig. 11). Let $cap(c_i), i = 1, 2$, denote the capacity of cut c_i. It is not hard to see that $cap(c_1)+cap(c_2)$ should be no less than $2d$ for the routing problem shown in Fig. 11, where the channel density is d. Usually, the channel is allowed to modelrately expand its width (the distance between its bottom and top) but not to expand its length.[d] It means that $cap(c_2)$ is assumed not to increase. With this assumption we obtain a nontrivial lower bound on the channel width $t \geq cap(c_1) \geq \frac{3d}{2}$, since $cap(c_2) = \frac{d}{2}$.

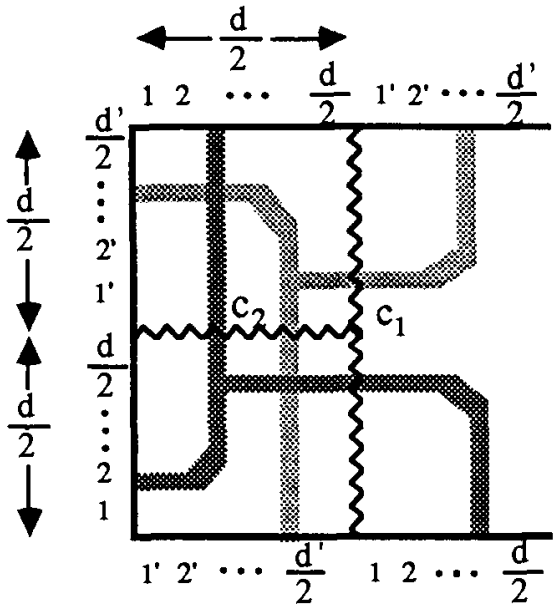


Fig. 11. A routing problem with terminals in three sides of a rectangle.

The example discussed above demonstrates the idea of using the properties of multiterminal nets and properly chosing a set of cross-sectional cuts to establish a lower bound. The mechanism explored here is totally different from that presented earlier in this paper where the way of using a grid point is classified. It provides us

[d]The bottom and top of a channel are usually the boundaries of circuit modules which have been fixed before the routing phase.

a different angle towards the understanding of the computational complexity of the problem. We can apply this approach to the establishment of lower bounds on the area of routing nets in the junction of channels. We define a junction as follows.

Definition 6. *Two rectangles are adjacent to each other, if they share a non-zero length of their boundaries. A collection of adjacent rectangles is referred to as a junction.*

Particularly, we deal with four types of junctions, namely, L-, S-, T-, and X-junctions, and we define the corresponding junction routing problems by combining the routing problems of several adjacent rectangles.

An *L-junction* $L(t_1, t_2, x_0, y_0)$ is an "L"-shaped region that is the union of the following four rectangular regions (see Fig. 12(a)):

$$\begin{aligned} L &= \{(x,y) : x < 0, -t_1 \leq y \leq 0\} \\ T &= \{(x,y) : 0 \leq x \leq t_2, 0 < y\} \\ J_1 &= \{(x,y) : 0 \leq x \leq x_0, -t_1 \leq y \leq 0\} \\ J_2 &= \{(x,y) : 0 \leq x \leq t_2, -y_0 \leq y \leq 0\} \end{aligned}$$

where $0 \leq x_0 \leq t_2$ (the distance between the origin and the vertical segment of the "dent") and $0 \leq y_0 \leq t_1$ (distance between the origin and the horizontal segment of the "dent") are the offset parameters predetermined by the positions of the surrounding circuit blocks. We refer to sets L and T as the *left* and *top* channels and to the set $J_1 \cup J_2$ as the *junction area*. Any shortest Manhattan path between $(0,0)$ and $(x_0, -y_0)$ is called a *bottleneck* of the junction area. A simple L-junction (one without a "dent") corresponds to $x_0 = t_2$ and $y_0 = t_1$. The S-, T-, and X-junctions can be defined similarly and are illustrated in Figs. 12(b), (c), and (d), respectively (for complete definitions see Ref. 10). Sets L, R, T, and B in Fig. 12 represent the left, right, top, and bottom channels, respectively.

Instead of present a detailed proof we in the following list the lower bounds on the area of routing nets in the above defined junction routing problems. (The complete proof can be found in Ref. 10.)

Theorem 5. *The existential lower bound for any general L-junction is* $t_1 + t_2 \geq \frac{3}{2}(d_1 + d_2)$, *where* t_1 *and* t_2 *are the widths, and* d_1 *and* d_2 *are the densities of the left and top channels of the L-junction, respectively.*

Theorem 6. *The existential lower bound for any general S-junction is* $t_1 + t_2 \geq \frac{3}{2}(d_1 + d_2)$, *where* t_1 *and* t_2 *are the widths, and* d_1 *and* d_2 *are the densities of the left and right channels of the S-junction, respectively.*

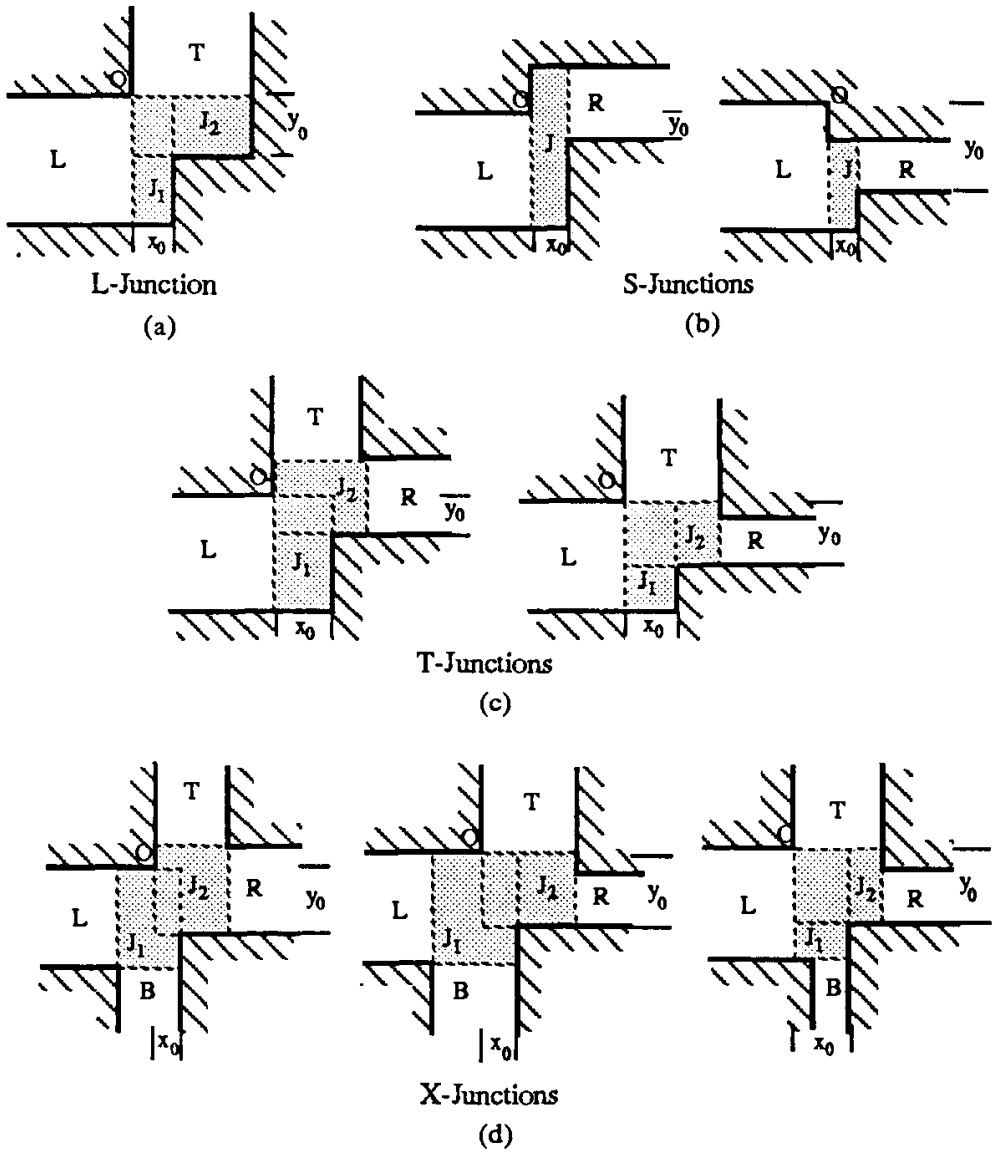


Fig. 12. Different kinds of junctions.

We denote the densities of the three (respectively, four) channels of a T-junction (respectively, an X-junction) in a clockwise sequence, starting at the left channel by d_i, for $1 \leq i \leq 3$ (respectively, for $1 \leq i \leq 4$), and their channel widths by t_i, for $1 \leq i \leq 3$ (respectively, for $1 \leq i \leq 4$).

Theorem 7. *The existential lower bound for any general T-junction is* $t_1 + t_2 + t_3 \geq \frac{3}{2}(d_1 + d_2) + d_3$.

Theorem 8. *The existential lower bound for an X-junction is* $t_1 + t_2 + t_3 + t_4 \geq \frac{3}{2}(d_1 + d_2) + d_3 + d_4$.

6. Discussion

We have shown new lower bounds on the channel widths for CRPs and SBRPs in the Manhattan model. These bounds are established by introducing the notion of "holes" and developing arguments on the supply and the demand of such a commodity inside a chosen region. Realizing that the hole is the grid point where only one nontrivial net is allowed to jog, we see that the more nets use grid points separately (without sharing) the stronger lower bound is. This happens only in the Manhattan model. For the knock-knee model, every grid point can be shared by two nets; therefore, the argument on the supply and the demand is not applicable. This explains why two-terminal CRPs can be routed optimally in the knock-knee model. While the knock-knee model produces a more compact routing, it has to use one or two more layers than the Manhattan model does (Refs. 1, 2, 3, 13, 14). The achieved saving in the area is at the expense of using additional layers.

Many heuristic routing algorithms use the "bench-mark" problems presented in Ref. 6 to test their performance. Most of those heuristic algorithms achieve satisfactory solutions by using one or two extra tracks more than required by the channel density (Refs. 9, 6, 16, 8, 15, 5, 17). A careful examination of those examples shows the fluxes of those CRPs are no more than three (Ref. 4). In fact, any channel window, which has its density equal to d_m, has the flux of no more than two; therefore, there is no channel window which requires the number of tracks to be more than $d_m + 2$ from our lower bound. In this sense, these CRPs are not "hard" to route because they do not require that many grid points be used by only one net. This may be the reason that most practical problems can be routed by heuristic algorithms with a channel width slightly larger than d_m. However, our new lower bound shows that the channel width cannot be separately bounded by either the channel density d_m or the flux f.

We have demonstrated that instead of exploring the way in which a single grid point is used the shape of routing regions and the property of multiterminal nets also contribute to the complexity of the problem. In fact, we have been able to establish nontrivial lower bounds on the routing area by using these properties. However, it is not clear how the property of multiterminal nets affects the complexity of the routing problem, without considering the shape of routing regions. A long-standing open problem is: What is a nontrivial lower bound on the channel width for multiterminal net CRPs? In addition to density and flux, what other factors contribute to the complexity of such routing problems? The best known upper bound for the multiterminal net CRP in the Manhattan model is $\frac{3d_m}{2} + 0(\sqrt{d_m} \log d_m) + 0(f)$ (Ref. 7). The best known lower bound, $t \geq \sqrt{d_m^2 + 4f^2 - 6} - 2$, presented in this paper, does not take any advantage of the property of multiterminal nets.

References

1. M. Brady and D. Brown, "An algorithm for three layer channel routing using restricted overlap", *Proc. of the 23rd Allerton Conf. on Communication, Control and Computing.* 1985, pp. 674–675.

2. B. Berger, M. Brady, D. Brown and F. T. Leighton, "Nearly optimal algorithms and bounds for multilayer channel routing", to appear in JACM, 1992.
3. B. S. Baker, S. N. Bhatt and F. T. Leighton, "An approximation algorithm for manhattan routing", *Proc. of the 15th Ann. ACM Symp. on Theory of Computing* 1983, pp. 477–486.
4. D. J. Brown and R. L. Rivest, "New lower bounds for channel width", *Proc. of CMU Conf. on VLSI* 1981, pp. 178–185.
5. M. A. Breuer, *Design Automation of Digital Systems*, (Prentice-Hall, Englewood cliffs, NJ, 1972).
6. D. Deutsch, "A dogleg channel router", *Proc. of 13th Design Automation Conf.*, 1976, pp. 425–433.
7. S. D. Gao and M. Kaufmann, "Channel routing of multiterminal nets", *Proc. of 28th Sym. on Foundations of Comp. Sc.*, 1987, pp. 316–325.
8. T. C. Hu and K. S. Ernest, *Theory and Concepts of Circuits Layoput*, 1985, pp. 3–17.
9. A. Hashimoto and J. Stevens, "Wire routing by optimizing channel assingment", *Proc. of 8th Design Automation Conf.*, 1971, pp. 214–224.
10. S. R. Maddila and D. Zhou, "Lower and upper bounds for the general junction routing problem", Technical Repot ACT-90, Coordinated Science Laboratory, University of Illinois, 1988.
11. S. R. Maddila and D. Zhou, "Routing in general junctions", *IEEE Trans. Computer-Aided Design* **8(11)** (1989) 1174–1184.
12. H. Okamura and P. D. Seymour, "Multicommodity flows in planar graphs", *J. Comb. Theory, Ser. B* **31** (1983) 75–81.
13. F. P. Preparata and W. Lipski, "Three layers are enough", *IEEE 23rd Symp. on Found. of Compt. Sci.* (1982) 350–357.
14. F. P. Preparata and W. Lipski, "Optimal three-layer channel routing", *IEEE Trans. Computers* **33** (1984) 427–437.
15. R. L. Rivest and C. M. Fiduccia, "A greedy channel router", *Proc. of Design Automation Conf.*, 1982, pp. 418–424.
16. T. G. Szymanski, "Dogleg channel routing is NP-complete", *IEEE Trans. Computer-Aided Design* **4(1)** (1985) 31–40.
17. T. Yoshimura and E. S. Kuh, "Efficient algorithms for channel routing", *IEEE Trans. Computer-Aided Design* **1(1)** (1982) 23–35.

Algorithmic Aspects of VLSI Layout
Eds. M. Sarrafzadeh and D. T. Lee
© World Scientific Publishing Co.

SWITCH-BOX ROUTING UNDER THE TWO-OVERLAP WIRING MODEL

TEOFILO F. GONZALEZ, SHASHISHEKHAR KURKI-GOWDARA
Department of Computer Science, University of California, Santa Barbara, CA 93106, USA

and

SI-QING ZHENG
Department of Computer Science, Louisiana State University Baton Rouge, LA 70803, USA

ABSTRACT

We present a new algorithm for the switch-box routing problem under the two-overlap wiring model. An instance of this problem is given by a rectangle R, and a set N of nets each formed by two terminal points located on the boundary of R. The objective is to interconnect the points in each net by a set of wires inside R on k layers that satisfy the constraints in the two-overlap wiring model. In this wiring model at most two wires may be assigned to the same track or column unit segment in all the layers. Given any two-overlap wirable instance, our algorithm adds at most two tracks and two columns and wires it in seven layers. The time complexity for our algorithm is $O(n)$ when the set of n terminal points is initially ordered.

Keywords: Switch-box routing, two-overlap, efficient algorithms, layer assignment.

1. Introduction

Channel routing (CR) and *switch-box routing* (SBR) are fundamental detailed routing problems in computer-aided design of VLSI layout systems. An SBR problem instance consists of a rectangle R and a set of *signal nets*, $N = \{n_1, n_2, \ldots, n_m\}$, where the n_i's are mutually disjoint sets of *terminal points* located on the boundary of rectangle R. The objective is to construct a wiring in R, whenever one exists. In the CR problem the terminal points are located on two opposite sides of R, and the objective is to construct a minimum *channel width* wiring.

The CR and SBR problems under the non-overlap wiring models such as the *Manhattan model* and the *knock-knee model* have been extensively studied. It is well known that the general CR and SBR problems under these two models are NP-complete.[1,2] Many heuristic algorithms for the Manhattan mode CR problems have been proposed.[3,4,5,6,7,8,9] Baker's *et al.* algorithm[3] is the only one known to have a provably good performance with respect to the channel width. There are no known algorithms to determine routability of the Manhattan mode SBR problem. The only

known Manhattan mode *SBR* algorithms are heuristic algorithms.[10] Several special Manhattan mode *CR* and *SBR* problems have been investigated. The *CR* and *SBR* problems are called *compatible*, if there is at most one terminal point along each grid line. The compatible *CR* can be solved efficiently under the Manhattan routing model using the algorithm developed by Hashimoto and Stevens.[11] Gonzalez and Zheng[12] developed an efficient algorithm for two-layer wiring of any two-terminal-net compatible *SBR* problem routable under the Manhattan model. The algorithm stretches the layout introducing a fixed number of tracks and columns.

Unlike Manhattan mode wire layouts, which can be wired in two layers, most knock-knee mode wire layouts require more than two layers; however, problem instances that cannot be routed under that model become routable under the knock-knee wiring model. The two-terminal net knock-knee mode *CR* problem can be solved efficiently by Frank's algorithm,[13] and improved algorithms have been presented in several papers.[14,15,16,17] Approximation algorithms for the multiterminal-net knock-knee mode *CR* problem have been investigated.[15,16,18] Gao and Kaufmann's algorithm[18] has the smallest approximation bound, and when incorporated with Weiners-Lummer's layer assignment strategy,[19] a three layer wiring can be generated. Knock-knee mode two-terminal net *SBR* algorithms had been developed.[13,20] The currently best knock-knee mode multiterminal-net *SBR* algorithms reduce the multiterminal net *SBR* problems into two-terminal net *SBR* problems by vertically and horizontally stretching the layout.[20,21] For multilayer wiring the fundamental result is an efficient algorithm to wire in four layers any non-overlapped wire layout.[22]

The non-overlap wiring models prohibit wire segments in different layers to overlap in parallel (i.e., in at least one unit segment). The reason for restricting wires from overlapping is the undesired capacitance introduced by the overlap of two wires. Current VLSI technology allows overlap of metal layers. As the ratio gates/chip increases, more layers are needed for the wiring. Therefore, wiring models that allow wire overlap are becoming more popular.

In the past few years, the *CR* and *SBR* problems under the wire overlap model received considerable attention. Several different wire overlap wiring models have been considered. These models differ in the restrictions on the length of wire overlaps, the number of layers in which wires can overlap, or the number of separation layers between any two overlapped wire segments. Most of the previous wire overlap mode routing algorithms are for the *CR* problem.[23,24,25,26,27,28,29,30,31] In contrast, the investigation on wire overlap mode *SBR* problem has been limited.[32]

In this paper we present a new algorithm for the two-overlap *SBR* problem. An instance of the *SBR* problem consists of a rectangle R and a set $N = \{n_1, n_2, \ldots, n_m\}$ of m two-terminal nets, each formed by two terminal points on the boundary of R. Every grid point on the boundary of R has at most one terminal point. The objective is to connect the two terminal points in each net by introducing a set of wires inside R in k layers that satisfy the constraints in the two-overlap wiring model. In this model there are at most two wire segments assigned to the same

track or column unit segment in all the layers. The best known two-overlap routing algorithms are given in Refs. 27, 32. The algorithm by Cong *et al.*[27] wires in four layers any multiterminal net *CR* problem using $\lceil d/2 \rceil + 2$ tracks. Kaufmann's algorithm[32] generates a nine-layer two-overlap wiring for any two-terminal net routable *SBR* problem. For multiterminal net *SBR* problems Kaufmann[32] developed an algorithm that generates nineteen-layer layouts. The wirings generated by Kaufmann's algorithms have the property that all the wire segments in each layer are either vertical or horizontal, but not both.

We present an algorithm that given any two-overlap routable *SBR* problem, adds at most two tracks and two columns, and wires it in seven layers. When the set of n terminal points is initially ordered, the time complexity for our algorithm is $O(n)$. The layouts generated by our algorithm have the property that all the wire segments in each layer are either vertical or horizontal, but not both. Most of the vias introduced by our algorithm join wires in adjacent layers. The rest of the vias join wires in adjacent horizontal layers, and are located along two columns. Our algorithm can be generalized to solve the multiterminal-net *SBR* problem by transforming each multiterminal net into several two-terminal nets and stretching the layout.[20,21]

2. Preliminaries

Rectangle R is partitioned by a uniform grid L with h tracks (horizontal lines) and w columns (vertical lines). The intersection of a track and a column, or a track or a column with R is referred to as a *grid point*. An *edge* is a horizontal or vertical line segment that joins two adjacent grid points, and a *grid edge* is an edge that does not overlap with R. Each grid point at the boundary of R (excluding the corners of R) contains at most one *terminal point*. Let $N = \{n_1, n_2, \ldots, n_m\}$ be a partition of the set of terminal points into two-element sets called *nets*. Figure 1 gives a problem instance with $h = w = 5$ and $m = 10$.

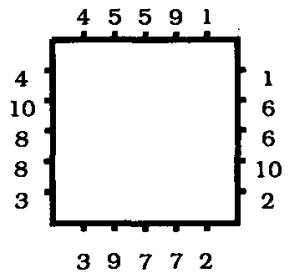


Fig. 1. A problem instance.

A k-layer wiring consists of k identical copies $L_1, L_2, \ldots, L_k$ of the grid L inside R, where L_i is above L_{i+1}. We refer to L_i as *layer i*. A k-layer layout for an *SBR* problem instance (denoted by $I = (R, N)$), is characterized by two mappings: wire

layout and layer assignment of the wire segments. A wire layout maps each net n_i in N to a wire W_i, which is a connected subgraph of L including both terminal points of net n_i. We refer to the set of wires $W = \{W_1, W_2, \ldots, W_m\}$ as *layout* of $I = (R, N)$. We say that W is an *l-layout* if at most l wire segments in W share any grid edge of L, and a *1-layout* is also called a *planar layout*. The layer assignment associates each wire segment on a grid edge e of L to its corresponding grid edge e' in a layer L_i, $1 \leq i \leq k$, in such a way that if two wire segments of wire W_i incident at a grid point p are assigned to layers i_1 and i_2 ($i_1 \leq i_2$), then no other wire segment of a wire W_j ($i \neq j$) incident at point p is assigned to a layer l, for all $i_1 \leq l \leq i_2$. When $i_1 < i_2$, we say that wire W_i has a *via* from layer i_1 to layer i_2 at point p. There could be several (disjoint) vias at the same grid point p when there are four or more layers, since more than one wire may be incident at grid point p. In what follows we refer to a horizontal (vertical) wire segment as an *hw*-segment (*vw*-segment).

A track or column is said to be *empty* if there are no terminal points at its intersection with R. Each track and column in a 2-layout consists of two *slots*, and wire segments may be assigned to either slot. The *corresponding track or column slot for a terminal point* are the track or column slots where the terminal point is located. The *corresponding track and/or column slots for a net* are the corresponding track and/or column slots for the terminal points of the net.

The bottom, top, left and right sides of R are labeled b, t, l and r, respectively. The set of nets N is partitioned into groups N_{rt}, N_{rb}, N_{lb}, N_{lt}, N_{tt}, N_{rr}, N_{bb}, N_{ll}, N_{tb}, and N_{lr}, where N_{xy}, for $x, y \in \{t, b, l, r\}$, represent the set of nets with one terminal located on side x of R and the other terminal located on side y of R. The net labeled i in Fig. 1 is in the i^{th} class defined above. A net is type *TB* if all its terminal points are located on the top and/or bottom sides of R (N_{tt}, N_{tb}, N_{bb}); it is type *LR* if all its terminal points are located on the left and/or right sides of R (N_{ll}, N_{lr}, N_{rr}); and it is type *neighbor* if its terminal points are located on two adjacent sides of R ($N_{rt}, N_{rb}, N_{lt}, N_{lb}$). Let $N_{TB} = N_{tt} \cup N_{tb} \cup N_{bb}$, $N_{LR} = N_{ll} \cup N_{lr} \cup N_{rr}$, and $N_{NN} = N_{rt} \cup N_{rb} \cup N_{lt} \cup N_{lb}$. A net is called a *trivial net* if both of its terminal points are located on the same track or column. The *natural bend number of a net* is the number of bends in a wire with least number of bends connecting the two points of the net. For example, the natural bend number of trivial nets is zero, for nets in N_{NN} is one and for non-trivial nets in N_{TB} is two. The *natural bend number of a problem instance is the sum of* the natural bend numbers of all its nets. The natural bend number of the problem instance in Fig. 1 is 16.

We say that net n_i *spans over the vertical (horizontal) line* l if net n_i contains a terminal point on each side of line l. The *density of line* l is the number of nets that span over line l. In Fig. 1, each vertical or horizontal non-grid line l that partitions R has density 3. The *vertical (horizontal) density*, $D_v(N)$ ($D_h(N)$) is defined as the maximum density of any vertical (horizontal) non-grid line. The problem instance given in Fig. 1 has $D_v(N) = 3$ and $D_h(N) = 3$. Obviously, if for

problem instance $I = (R, N)$ either $D_v(N) > 2h$ or $D_h(N) > 2w$, then there is no two-overlap wiring of the nets N in R. Therefore, necessary conditions for the existence of a wiring of N inside R are $D_v(N) \leq 2h$ and $D_h(N) \leq 2w$. It is not known whether or not these conditions are sufficient for the existence of a wiring of N in R. In what follows we "corrupt" our notation and say that a problem instance is *routable* if $D_v(N) \leq 2h$ and $D_h(N) \leq 2w$. Note that under this corrupted notation we might be calling some problem instances routable even though they are unroutable. It is interesting to note that any problem instance with $h = w$ is routable. This fact follows from Lemma 1 where we establish a relationship between h, w, D_v and D_h.

Lemma 1. *For problem instance $I = (R, N)$,*

(i) if $h \leq w$ then $D_h(N) \leq 2w$;
(ii) if $h \geq w$ then $D_v(N) \leq 2h$; and
(iii) if $h = w$ then $D_h(N) \leq 2w$ and $D_v(N) \leq 2h$.

Proof. Since the proof of the three parts is similar, we only prove (i). The only nets that can contribute to $D_h(N)$ are the nets in N_{tb}, N_{LR}, and N_{NN}. Each of these nets contributes at most one to $D_h(N)$. Therefore, $D_h(N) \leq |N_{tb}| + |N_{LR}| + |N_{NN}|$. Since each of these nets consist of two terminal points located on the boundary of R, there is at most one terminal point at each grid point in R (excluding the corners of R), and $h \leq w$, it must be that $2|N_{tb}| + 2|N_{LR}| + 2|N_{NN}| \leq 2h + 2w \leq 4w$. Therefore, $D_h(N) \leq 2w$. □

For problem instances that are not routable, one may introduce additional tracks or columns until it becomes routable. Hereafter we assume without loss of generality that $h \leq w$. Given a routable problem instance $I = (R, N)$, our algorithm extends the rectangle with at most two additional tracks and two columns, and constructs a 7-layer two-overlap wiring. Our procedure for constructing 2-layouts is similar to the one that constructs a planar layout for the *SBR* compatible net problem.[12] However, the layer assignment strategy in this paper is much more complex, and the underlying wiring models are quite different. We should also point out that the 2-layout constructed by our routing procedure, and our layer assignment strategy are different from the ones given in Ref. 32. The wires in the 2-layouts constructed by the algorithm in Ref. 32 do not have a minimum number of bends. In our 2-layouts at least two-thirds of the nets are connected by wires with a number of bends equal to the natural bend number of the net. The remaining nets (at most one-third) are connected by wires with a number of bends that exceeds by two the natural bend number of the net. There are problem instances (Fig. 2) for which there does not exist a 2-layout in which more than 66% of the nets are connected by wires with a number of bends equal to the natural bend number of the net. We should point out that constructing a 2-layout is simple, the difficulty arises in constructing a 2-layout that is seven-layer wirable. It is worth noting that four layers are normally

required for wiring a planar layout.[22] One would expect that for 2-layouts eight layers are required. In general this might be true, but for the specific 2-layouts that we construct, only seven layers are required.

3. Routing Algorithm

We present procedure *ROUTE* that for any two-terminal-net two-overlap routable *SBR* problem instance constructs a 2-layout. Because of some special properties the 2-layouts are wirable in only seven layers (Sec. 4). Procedure *ROUTE* constructs a 2-layout by executing the following procedures: extension (*EXT*), trivial net routing (*ROUTE-TN*), neighbor net routing (*ROUTE-NN*), routing *LR* nets and a subset of *TB* nets (*ROUTE-LRTB* and *ROUTE-LR*), and routing the remaining problem (*ROUTE-REM*). First we outline the basic steps of our procedures and justify some of their critical steps. Then we formally define our procedures.

The problem instance is given by $I = (R, N)$, where R is a rectangle with h (w) interior tracks (columns) and N is the set of m two-terminal nets. Remember that we assume instance I is routable and $h \leq w$. In procedure *EXT*, one additional empty track is introduced next to the top boundary. The slots in this track are labeled t and t'. All of the trivial nets are routed (*ROUTE-TN*) in the obvious way in their corresponding track or column slots. In the neighbor net routing step (*ROUTE-NN*), each neighbor net is connected by an "L" shaped wire in its corresponding track and column slots.

Procedure *ROUTE-LRTB* routes a subset of nets that includes either all *TB*, or all *LR* nets. At each iteration in *ROUTE-LRTB*, two unrouted nets are selected, a *TB* net and an *LR* net. Then the two nets are routed in their corresponding track and column slots, and in certain cases in the track slot labeled t. Procedure *ROUTE-LRTB* terminates when either all *LR* or all *TB* nets have been routed. Rectangle R is vertically stretched by introducing two new grid lines without terminal points nor vertical wires, and in some cases R is also stretched horizontally. In procedure *ROUTE-LR*, each unrouted *LR* net (if any) is routed in a distinct column slot and in its corresponding track slots.

At this point one may expect that the routing of the remaining nets is straightforward. Unfortunately, this is not the case. Consider the problem instance in Fig. 2(a). After routing nets 1 through 4, it is impossible to wire both nets a and b. However, the problem instance can be wired by re-routing a net and introducing two additional bends (see the following procedure). By introducing additional nets, as suggested by Fig. 2(b), we can generate problem instances in which at most 66% of the nets are connected by wires with a number of bends equal to the natural bend number. Our algorithm must backtrack before a 2-layout may be generated for all possible problem instances. The reason why we do not backtrack to the beginning of the algorithm is that the resulting problem at this point is in general significantly

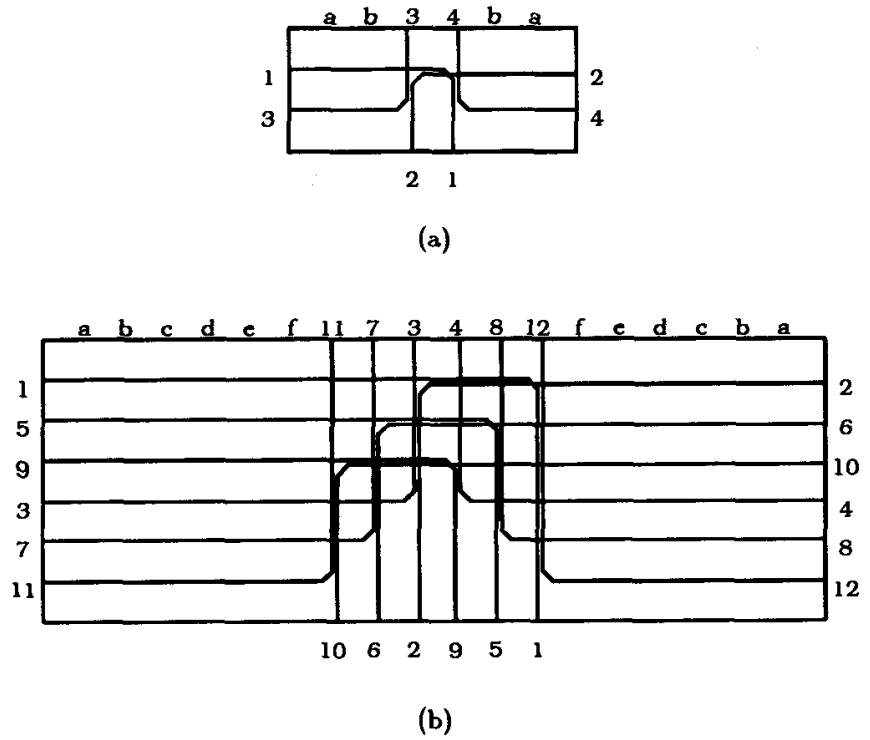


Fig. 2. Problem instances for which we must backtrack.

simpler than the original problem. The following procedure removes a set of wires, and then generates a 2-layout.

If all nets have been routed, then procedure *ROUTE* terminates; otherwise, we execute procedure *ROUTE-REM*. The procedure begins by removing wire segments inside a rectangle that includes all tracks in R, and a set of adjacent columns in R. The resulting problem is referred to by $I' = (R', N')$. The set of nets N' is partitioned into *runs*, such that the vertical density of each *run* is one, and the total number of *runs* is equal to the vertical density of the unrouted nets. At each iteration we route all nets in one or two *runs*. The 2-layout for this new problem plus the previously introduced wire segments (that were not deleted at the beginning of this procedure) form the 2-layout which is the output of procedure *ROUTE*.

Let us explain in detail algorithm *ROUTE* when invoked with problem instance $I = (R, N)$. We use the example given in Fig. 3 to illustrate our algorithm. By convention $h \leq w$ and instance I is routable, i.e., $D_h(N) \leq 2w$ and $D_v(N) \leq 2h$. In Subsection 3.1, we introduce our notation, and in Subsection 3.2 we present our procedure for routing trivial and neighbor nets. Procedures *ROUTE-LRTB* and *ROUTE-LR* are explained in Subsection 3.3, and Subsection 3.4 explains procedure *ROUTE-REM*. Finally, in Subsection 3.5 we establish our main result for this Section.

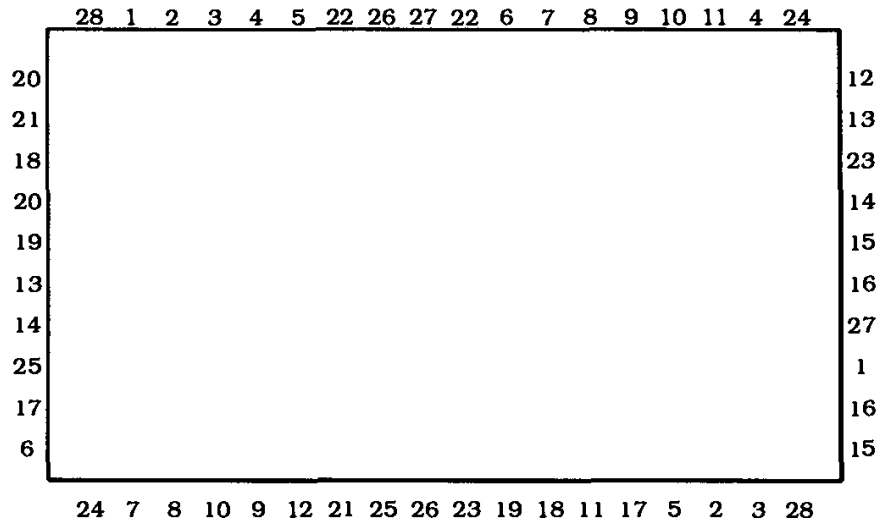


Fig. 3. Problem instance $I = (R, N)$.

3.1. Notation

Procedure *EXT* introduces an empty track between the top side of R and the topmost track. For convenience we shall not modify the value of h, so actually, after this iteration the number of tracks is not h, but $h + 1$. The track slots in this new empty track are labeled t and t'. All track and column slots are labeled *available* for routing, except for track slot t which is labeled *reserved*, and track slot t' which is labeled *unavailable* for routing. We uniquely *match* each terminal point from an unrouted net with one of its corresponding track or column slots that is *available*. For trivial nets, only one of its terminal points is *matched* with a track or column slot. From the initial conditions we know that a matching is always possible. Hereafter, an *available* track or column slot is said to be either *matched* with a terminal point from an unrouted net, or *not-matched*. Once a set of nets is routed in an *available* track or column slot, the slot is relabeled *unavailable* or *reserved*.

Let *mats* (*macs*) represent the number of *matched available* track (column) slots, and let *nmats* (*nmacs*) the number of *not-matched available* track (column) slots. A trivial net whose terminal points are located on the top and bottom side of R is called a *c-trivial* net. Let T_{TB} be the set of c-trivial nets, let $u(T_{TB})$ be the set of unrouted c-trivial nets, and let $|u(T_{TB})|$ denote the number of unrouted c-trivial nets. In Lemma 2 we establish that some important properties hold just after labeling the tracks. These invariants are needed to establish correctness of our algorithm. For example, inequality (i) below will be used to show that there is enough space to route the remaining set of unrouted nets.

Lemma 2. *Just after the track and column slots are labeled, the following statements hold.*

(i) $mats + |u(T_{TB})| \leq macs + 2nmacs$.

(ii) *Each terminal point from the unrouted LR (TB) nets is uniquely matched with one of its corresponding available track (column) slots, except for terminal points from trivial nets, in which case, only one of the terminal points is matched.*

(iii) *Track slot t does not contain wires.*

(iv) *Each non-grid vertical line intersects the hw-segments from each net at most once.*

Proof. The proof for (ii) is straightforward, and (iii) and (iv) hold because there are no wires. Let us now prove (i). Since each neighbor and c-trivial net is *matched* with one column slot, each non-trivial TB net is *matched* with two column slots, and $T_{TB} \subseteq N_{TB}$, we know that $macs = |N_{NN}| + 2|N_{TB}| - |T_{TB}|$. By definition, $nmacs = 2w - macs$. Combining the two expressions we know that $macs + 2nmacs = 4w - |N_{NN}| - 2|N_{TB}| + |T_{TB}|$. Since each neighbor net has one terminal point on the top or bottom side of R, each TB net has two terminals on the top and bottom side of R, and the total number of terminal points located on the top and bottom side of R is at most $2w$, we know that $|N_{NN}| + 2|N_{TB}| \leq 2w$. Substituting in the above expression we know that $macs + 2nmacs \geq 2w + |T_{TB}|$. Since $w \geq h$ and $2h \geq mats$, we know that $macs + 2nmacs \geq mats + |T_{TB}|$.

Just after the track and column slots are labeled, $|T_{TB}|$ is equal to $|u(T_{TB})|$. Substituting in the above inequality, we obtain (i). Therefore, (i)–(iv) hold just after the track and column slots are labeled. □

3.2. Trivial and Neighbor Net Routing

Each trivial net is routed by *ROUTE-TN* in the obvious way in the track or column slot *matched* with one of its terminal points. Each track or column slot involved in the routing of the trivial net is labeled *unavailable*. Therefore, in each iteration of *ROUTE-TN* either one net in $u(T_{TB})$ is deleted or *mats* decreases by one, and *macs* decreases by at most one. The value of $u(T_{TB})$ is zero when procedure *ROUTE-TN* terminates. We claim that (i)–(iv) in Lemma 3 hold when procedure *ROUTE-TN* terminates.

Lemma 3. *Statement (i)–(iv) in Lemma 2 hold when procedure ROUTE-TN terminates.*

Proof. The proof is omitted since it is straightforward. □

Next we apply procedure *ROUTE-NN*, to route the set of neighbor nets. At each iteration we select a neighbor net, and route it by an "L" shaped wire. Then, we label the track and column slots *matched* with the net's terminal points *unavailable*. Therefore, just after each iteration *mats* and *macs* decrease by one. Figure 4 shows

our example after procedures *EXT*, *ROUTE-TN*, and *ROUTE-NN* are executed. Note that in this example there are no trivial nets.

Lemma 4. *Statement (i)–(iv) in Lemma 2 hold just after procedure ROUTE-NN terminates.*

Proof. The proof is omitted since it is straightforward. □

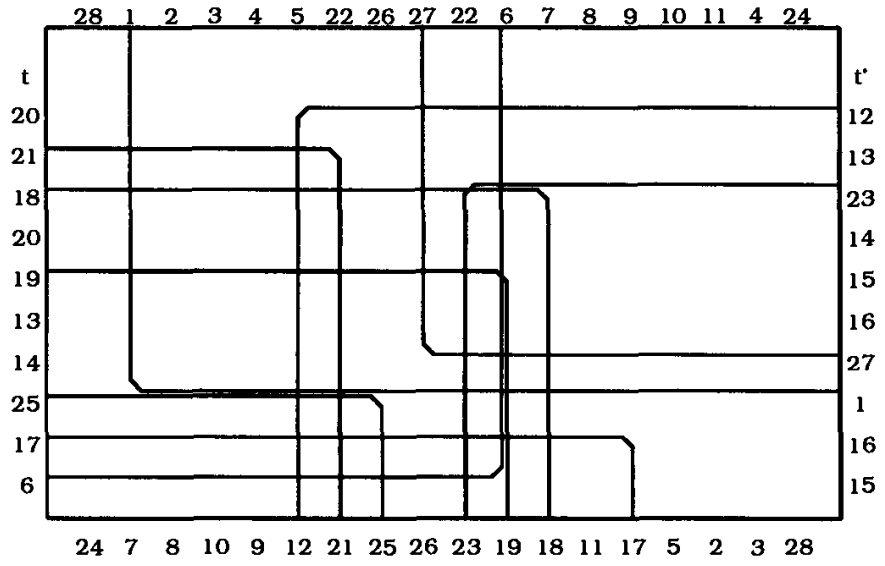


Fig. 4. Our example after introducing the new track and routing all neighbor nets.

3.3. Procedures ROUTE-LRTB and ROUTE-LR

We apply procedure *ROUTE-LRTB* which iterates until either all *LR* or all *TB* nets have been routed. Then, procedure *ROUTE-LR* routes all the unrouted *LR* nets (if any). Let us formally define each of these procedures.

Procedure *ROUTE-LRTB* iterates until all *LR* or all *TB* nets have been routed. At the beginning of the first iteration we define a window G inside R, and at the end of each iteration G is updated. Window G is the smallest rectangle inside R such that all terminal points from the unrouted *LR* (*TB*) nets can be horizontally (vertically) projected to its boundary. Project terminal points from unrouted *LR* and *TB* nets to their nearest points in G. We say that a corner of G is ***doubly-covered*** if two terminal points were projected to it. Let *left (right)* represent the x-coordinate value of the left (right) boundary of G.

Two unrouted nets (one from *LR* and one from *TB*) are routed at a time. The specific nets to be routed at each iteration depend on whether there are ***doubly-covered*** corners in G.

Case 1: There is a ***doubly-covered*** corner in G.

Assume that the *doubly-covered* corner is located on the bottom-right corner of G and that t is above G; the other cases are treated similarly. If the rightmost *available* column is on the right boundary of the rectangle G, then Fig. 5 shows the actions performed in this case. Our "visual" notation is defined as follows. A dotted line represents a track or column slot that is marked *unavailable* at this step; the dashed line represents the *reserved* track slot t; and a solid thick line represents a net routed at this step. In this case a knock-knee is introduced. Figure 5 does not show all the wires introduced for the two nets (the missing part of the wires only occupy part of the dotted track or column slot up to where it ends). On the other hand, if the rightmost *available* column is to the right of rectangle G, then the LR net is routed in that column. As a result of this, that column is labeled *unavailable*, and the column on the right boundary of G is labeled *not-matched available*.

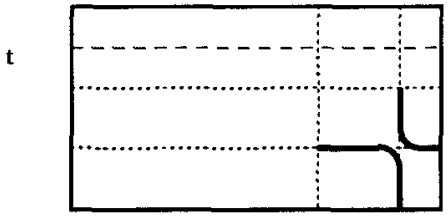


Fig. 5. Wires introduced for Case 1.

Case 2: There are no *doubly-covered* corners in G.

The two Subcases 2(a) and 2(b) that arise are shown in Figs. 6(a) and (b), respectively. The line segments intersecting the boundary of R show the locations where the terminal points from unrouted nets were projected to the corners of G.

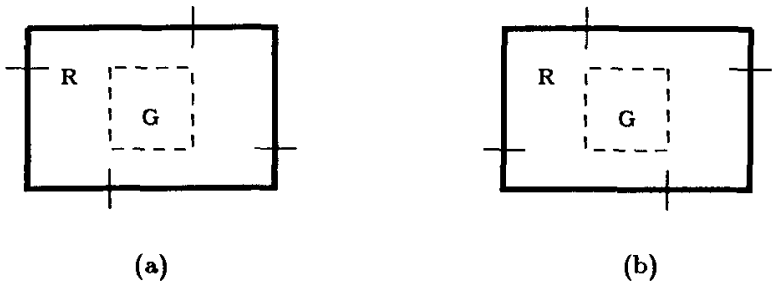


Fig. 6. There is no doubly-covered corner in G.

Assume without loss of generality that t is above G. If Case (a) ((b)) applies and the rightmost (leftmost) *available* column is on the right (left) boundary of G, then the wire segments introduced in this case are shown in Fig. 7(a) ((b)) and corresponds to the case shown in Fig. 6(a) ((b)). After the iteration the dotted-dashed line represents the new *reserved* track slot t, and the previous track slot t becomes *unavailable*. On the other hand, if Case (a) ((b)) applies and the rightmost

(leftmost) *available* column is to the right (left) of rectangle G, then the LR net is routed in that column. As a result of this, that column is labeled *unavailable*, and the column on the right (left) boundary of G is labeled *not-matched available*.

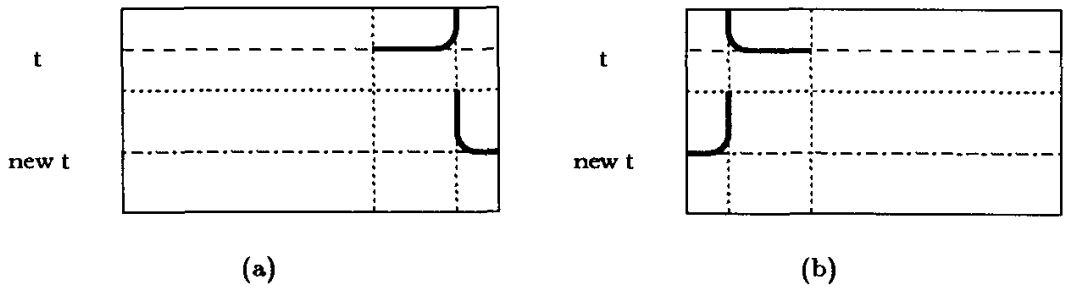


Fig. 7. Wires introduced in Case 2 when t is above G.

Clearly, at each iteration *mats* and *macs* decrease by exactly two. At the end of the iteration, the track labeled t is not *matched* with a terminal point from an unrouted net, and track t does not have a wire from column *left* to column *right* (*left* and *right* are defined with respect to the G at the end of the iteration). The above process is repeated as long as there are unrouted LR and TB nets. Procedure *ROUTE-LRTB* terminates when there remain no unrouted TB or LR nets.

Figure 8 shows the wires introduced in our example by procedure *ROUTE-LRTB*, but for clarity does not show the wires introduced during the previous steps. In all our figures we use a solid dot on the boundary of R to indicate the location of an *unavailable* track or column slot (in this case the track or column slot was *unavailable* when procedure *ROUTE-LRTB* began). Similarly, an X represents a *not-matched available* track or column slot. In our example, procedure *ROUTE-LRTB* performs the following steps:

1. Rectangle G is defined. Nets 28 and 15 are wired from the bottom-right corner of G (Case 1), and G is updated.
2. Nets 24 and 16 are wired (Case 2(a)). Track slot t is now the track slot *matched* with the bottommost terminal point from net 16, and G is updated.
3. Nets 7 and 14 are wired from the bottom-left corner of G (Case 1), and G is updated.
4. Nets 2 and 20 are wired from the top-left corner of G (Case 1), and G is updated.
5. Nets 8 and 13 are wired from the bottom-left corner of G (Case 1), and G is updated.

In Lemma 5, we establish that five statements (including statements (i)–(iv) of Lemma 4) hold at the beginning of each iteration of procedure *ROUTE-LRTB*, and when procedure *ROUTE-LRTB* terminates.

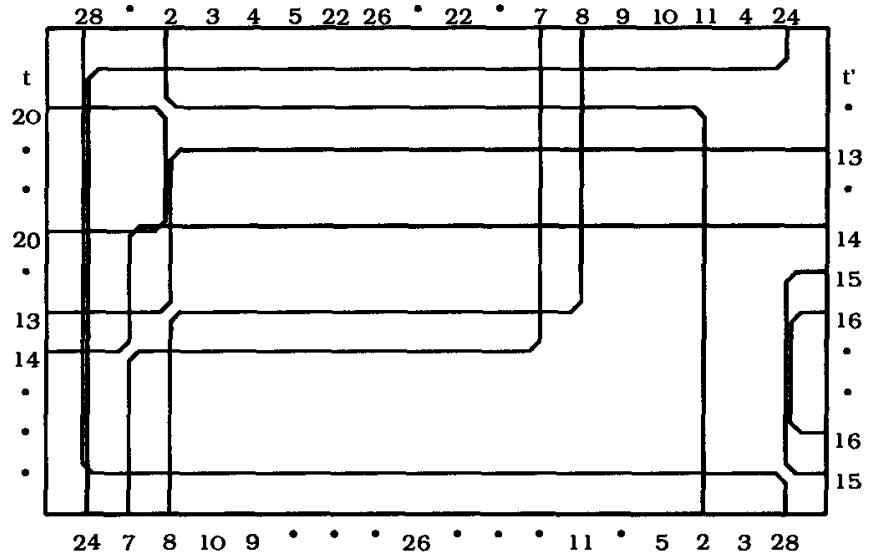


Fig. 8. Wires introduced by procedure *ROUTE-LRTB*.

Lemma 5. *Statements (i)–(v) hold at the beginning of each iteration of ROUTE-LRTB, and when procedure ROUTE-LRTB terminates.*

- *(i)* $mats \leq macs + 2nmacs$.
- *(ii) Each terminal point from unrouted LR (TB) nets is uniquely matched with one of its corresponding available track (column) slot.*
- *(iii) Track slot t is not in a track inside G and it does not contain hw-segments from column left to column right.*
- *(iv) Each vertical line that intersects the open interval from column left to column right intersects the hw-segments from each net at most once.*
- *(v) All the terminal points from unrouted LR (TB) nets can be horizontally (vertically) projected to the boundary of G.*

Proof. By Lemma 4 and the definition of G, we know that statements (i)–(v) hold at the beginning of procedure *ROUTE-LRTB*. It is simple to show that if (i)–(v) hold at the beginning of an iteration, they also hold at the end of the iteration. Thus, the lemma follows by induction. □

Figure 9 shows all the wires introduced in our example by procedure *ROUTE-LRTB*. If all nets have been routed, then let *left* (*right*) be the left (right) boundary of last rectangle G defined in *ROUTE-LRTB*; unless no rectangle G was ever defined, in which case *left* and *right* are adjacent columns in the middle of rectangle R. If there remain only *TB* unrouted nets, then let *left* (*right*) denote the leftmost (rightmost) column where a terminal point from an unrouted *TB* net is located, unless such a column contains the *vw*-segment of an *LR* net, in which case, *left* (*right*) is set to the column immediately to its right (left). On the other hand, if there remain only *LR* unrouted nets, let *left* (*right*) be the leftmost (rightmost)

column that is *available*. Rectangle R is stretched by introducing two vertical grid lines (without terminal points nor vertical wires) between columns *left-1* (*right*) and *left* (*right+1*). The horizontal wires intersecting these columns are stretched and preserve the previous connectivity. The values of *left* and *right* are updated to point to these new columns. Rectangle R' includes all tracks, and columns $left, left+1, \ldots, right$ in R. In Sec. 4 it will be evident why we have introduced the two additional columns. If all the nets have been routed then we proceed to the layer assignment phase (Sec. 4). On the other hand, if all the *LR* nets have been routed, we apply procedure *ROUTE-REM*; otherwise we apply procedure *ROUTE-LR*. These procedures are defined below.

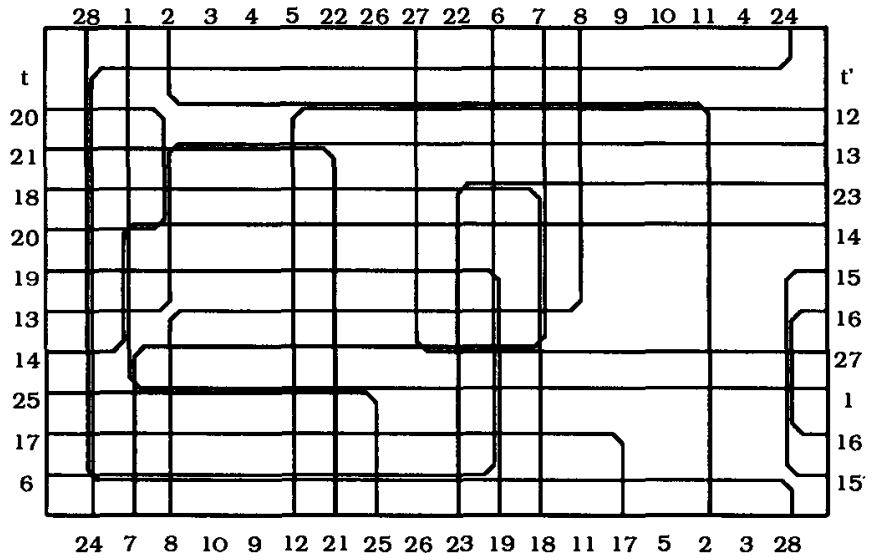


Fig. 9. All wires introduced after *ROUTE-LR*.

In procedure *ROUTE-LR* we route all the remaining unrouted *LR* nets. In our example, all *LR* nets have been routed (see Fig. 8), so procedure *ROUTE-LR* does not introduce new wires. Let us now consider the case when there remain unrouted *LR* nets. Each unrouted *LR* net is routed in the two track slots *matched* with its terminal points, and in a *not-matched available* column slot. An N_{ll} (N_{lr} or N_{rr}) net is routed in the leftmost (rightmost) *not-matched available* column slot. By Lemma 5(i) we know that at least one such column slot exists for each of the remaining unrouted *LR* nets. The track and column slots involved in the routing of each of these nets are labeled *unavailable*. At the end of procedure *ROUTE-LR*, rectangle G is redefined as a single grid point inside the previous G. In Lemma 6 we establish some important properties of the wiring generated by procedure *ROUTE-LR*.

Lemma 6. *Statements (i)-(v) hold at the beginning of procedure ROUTE-LR, and just after procedure ROUTE-LR terminates.*

Proof. By Lemma 5 it follows that (i)–(v) hold at the beginning of procedure *ROUTE-LR*. From the above discussion, we know that if (i)–(v) hold at the beginning of procedure *ROUTE-LR*, then they also hold when procedure *ROUTE-LR* terminates. □

3.4. Procedure ROUTE-REM

As mentioned before, we must backtrack in order to be able to route the remaining nets. Let us now define the remaining problem, which is considerably simpler to route than the original problem. From the previous procedures we know that the only unrouted nets are *TB* nets. The remaining problem, which we shall refer to as $I' = (R', N')$, is defined by deleting some of the previously introduced wire segments. Specifically, all the *hw*-segments segments located on a track from column *left* to column *right*, and all the *vw*-segments from column *left*+1 to column *right*-1 are deleted. In our example, columns *left*+1 and *right*-1 contain terminal points of routed nets. In general, this may not be true. We claim that at most one terminal point in this column could belong to an unrouted *TB* net. In this case, we introduce a new horizontal track at the bottom of the rectangle, and t' will be one of the track slots in the bottom track. The other slot in the bottom track, and the old track slot t', are used for wire segments that connect the unrouted terminal points to the boundary of R', without introducing any segments inside R'. Figure 10 shows the remaining wires after applying this step to the 2-layout given in Fig. 9.

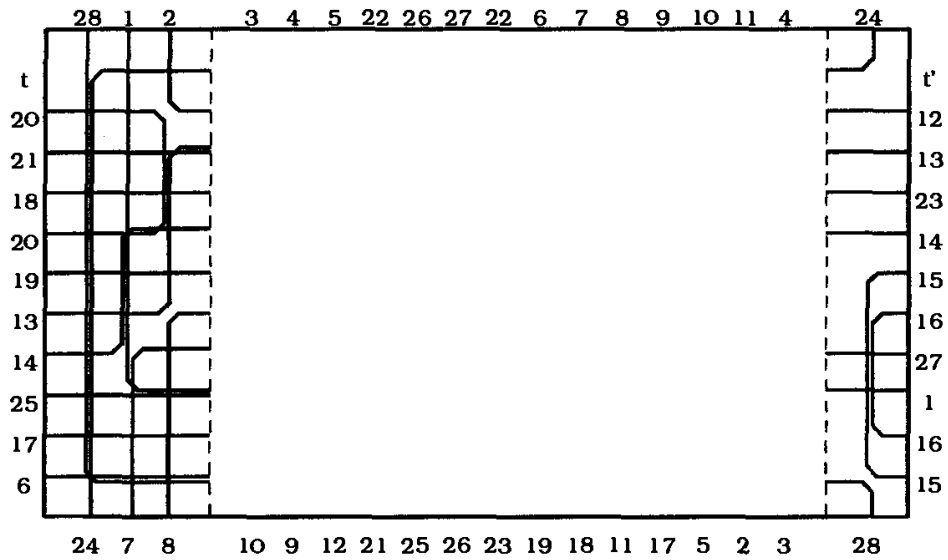


Fig. 10. Remaining wires.

It is important to remember that rectangle R' includes all tracks in R, and the columns $left$, $left+1, \ldots, right$. The terminal points in R' consist of all the terminal points from previously unrouted nets plus new terminal points located at the intersection of the boundary of R' with a deleted wire (these points of intersection

are labeled with the name of the corresponding net). For R', we have a set N' of two terminal nets. Figure 11 shows the rectangle R' for our example. Note that there may be more than one terminal point at the intersection of a track with the left or right boundary of R'. However, if this is the case, at least one of them belongs to a trivial net, i.e., the corresponding point on the opposite side of R' has the same label, and either the track has two trivial nets, or one trivial net and at most another terminal point that belongs to an unrouted net in N'. It is simple to show that the set N' of nets can be partitioned into N'_{TB}, N'_{NN}, and trivial N'_{LR} nets. This simplifies considerably the routing problem, and is the main reason why one should not eliminate the first few steps in our procedure. Note that each of the nets of N' may not be of the same type as in N. Figure 11 shows the problem $I' = (R', N')$ generated for our example, and Lemma 7 establishes that I' is routable.

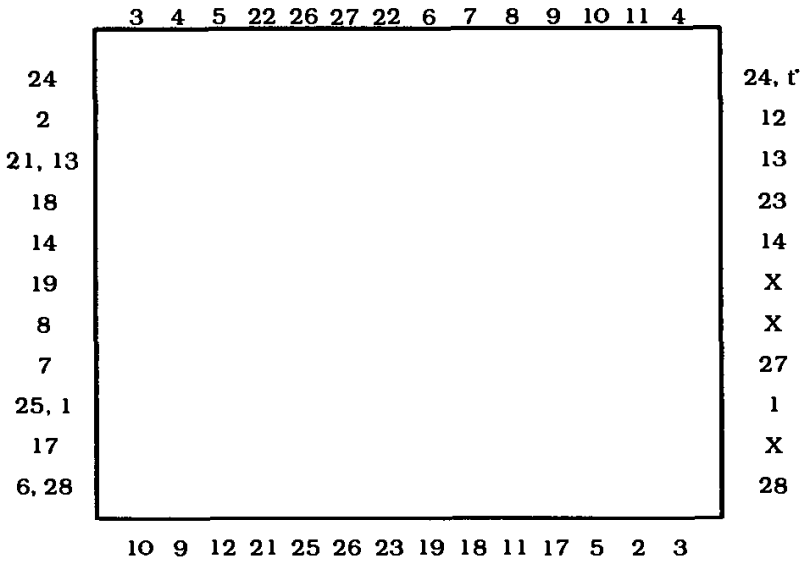


Fig. 11. Problem instance I' for our example.

Lemma 7. *Problem instance I' constructed at the beginning of procedure ROUTE-REM is routable, i.e. the horizontal and vertical densities of problem instance I' satisfy the necessary conditions for routability.*

Proof. The proof follows from Lemma 6, and the discussion in the previous paragraph. □

The final layout consists of the remaining wires in R (those not deleted at the beginning of this step) plus the wires introduced inside R' by *ROUTE-REM*. The remaining problem consists of introducing a set of wires inside R' to connect the nets in N'. This is a restricted version of the original problem, with the possible exception that there could be two terminal points on a track at the intersection

with the left or right boundary of R'. However, at least one of those two terminal points belongs to a trivial net in N'_{LR}. All the column and track slots are labeled *available*, except for track slot t' which is labeled *reserved*. Each terminal point from an unrouted net (except for trivial N'_{LR} nets in which case only one of its terminal points) is uniquely *matched* to one of its corresponding track or column slots. It is simple to show that a matching is always possible. Hereafter, an *available* track or column slot is said to be either *matched* with a terminal point from an unrouted net or *not-matched*.

Procedure *ROUTE-REM* begins by routing the trivial N'_{LR} and the trivial N'_{TB} nets. Each of these nets is routed in the obvious way in the *available* track or column slot *matched* with one of its terminal points, and such a slot is relabeled *unavailable*. Figure 12 shows our example after routing trivial nets. In our example there are no trivial N'_{TB} nets.

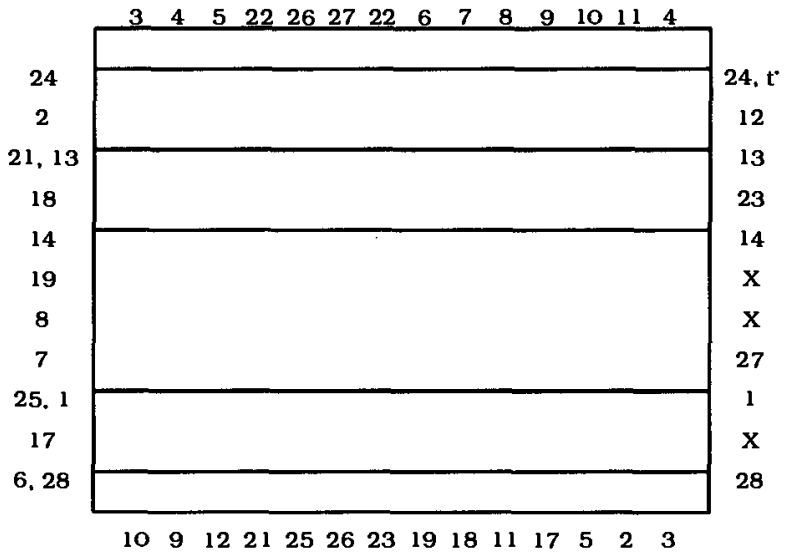


Fig. 12. Wiring trivial nets.

Note that each time a trivial N'_{LR} (N'_{TB}) net is routed, the vertical (horizontal) density of the unrouted nets, and the number of *matched available* track (columns) slots, decreases by one. Therefore, if problem instance I' was routable, then the resulting problem is also routable. The resulting problem for our example is shown in Fig. 13. Remember that a solid dot indicates the location of an *unavailable* track or column slot, and an X indicates the location of a *not-matched available* track or column slot.

Procedure *ROUTE-REM* groups the unrouted nets into sets called *runs*, which are constructed as follows. Initially, all unrouted nets are marked as *unvisited*. The nets in the first *run* are identified first, then the ones in the second *run*, and so forth. We identify the nets in the i^{th} *run* as follows. The first net in the i^{th} *run* is an unvisited net whose leftmost terminal point has the smallest x-coordinate value

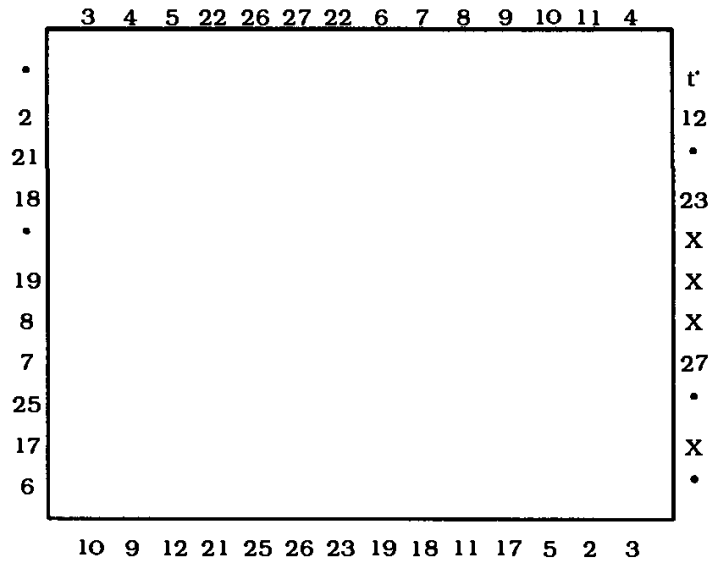


Fig. 13. Resulting problem after wiring trivial nets.

(in case of ties select any one of them). For $k = 2, 3, \ldots$, the k^{th} net in the i^{th} *run* is an unvisited net whose leftmost terminal has the smallest x-coordinate value, and such value is greater than or equal to the x-coordinate value of the rightmost terminal of the $(k-1)$ net in the i^{th} *run*. Once all the nets in the i^{th} *run* have been identified, we mark them as visited and proceed to find the $(i+1)$st *run*. This process is repeated until all unrouted nets are marked visited. Lemma 8 establishes an important property of *runs*.

Lemma 8. *The number of runs identified by procedure ROUTE-REM is less than or equal to the number of available track slots.*

Proof. The proof follows from Lemma 7, the fact that each *run* has density one, and the fact that the vertical density of the *runs* is equal to the number of *runs*. □

Depending on the number of neighbor nets in a *run*, the *run* is called a 0_n *run*, a 1_n *run* or a 2_n *run*. In our example, procedure *ROUTE-REM* identifies the *runs* given in Table 1. Since our example is small, most *runs* consist of only one net; however, in general, *runs* may contain several nets.

Procedure *ROUTE-REM* then applies the following three rules (in any order) until all *runs* are routed, or the rules do not apply. Later on we show how to route the remaining *runs* (if any).

(a) An unrouted 1_n *run* is routed in the track slot *matched* with its neighbor net, and in the column slots *matched* with the nets in the *run*. These column and track slots are labeled *unavailable*.

Table 1. Runs for our example.

Type	Runs
0_n	3, 4, 5, 9, 10
1_n	2, 6, 7, 8, 12, 17, 18-11, 19
2_n	21-22-23, 25-26-27

(b) If the two track slots *matched* with the neighbor nets in an unrouted 2_n *run* belong to the same track, then route the 2_n *run* in one of these track slots and in the column slots *matched* with the nets in the *run*. Label all of these column and track slots *unavailable*. The unused track slot *matched* with the neighbor nets in the *run* becomes a *not-matched* (but, still *available*) track slot.

(c) When there is a *not-matched available* track slot, and an unrouted 0_n *run*, route an unrouted 0_n *run* in the *not-matched available* track slot, and in the column slots *matched* with the nets in the *run*. Label all of these column and track slots *unavailable*.

The wiring generated by applying the above three rules to our example is given in Fig. 14. Note that rule (b) is not applied in this example.

After procedure *ROUTE-REM* applies these three rules as many times as possible, the only remaining unrouted *runs* (if any) are 0_n and 2_n *runs*. The unrouted nets in our example are given in Fig. 15. We claim that the remaining unrouted *runs* satisfy the conditions of Lemma 9.

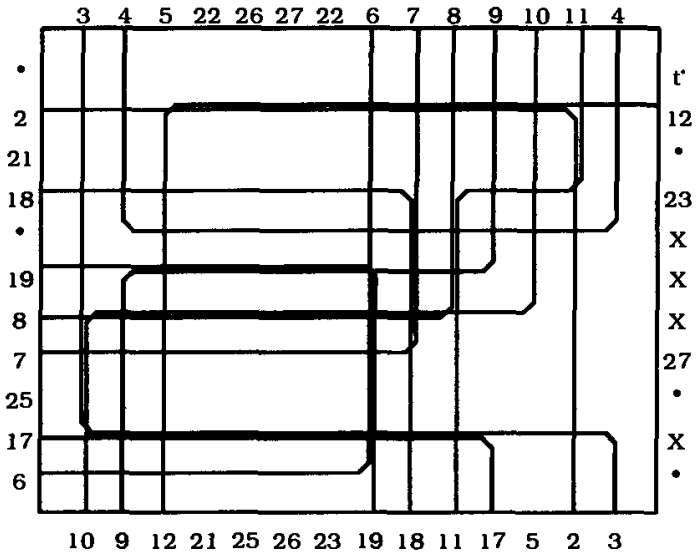


Fig. 14. Wiring of the *runs*.

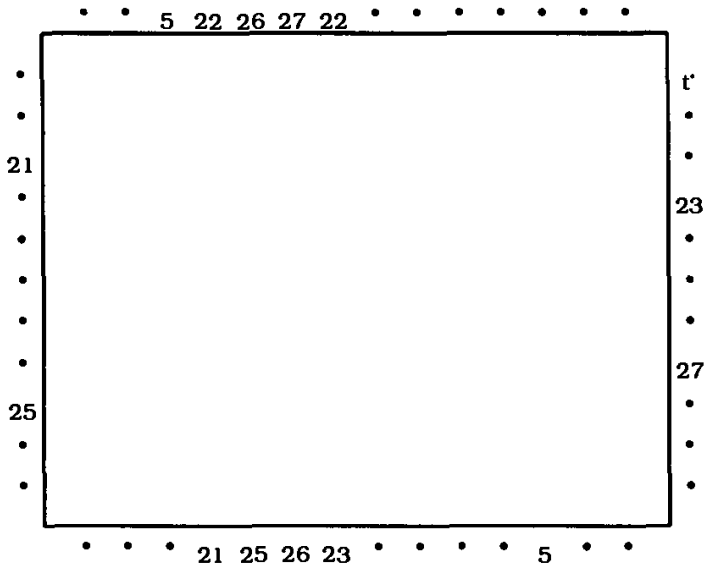


Fig. 15. Wiring of the *runs*.

Lemma 9.

1. *After every application of rules (a)-(c), the number of unrouted runs is less than or equal to the number of available track slots.*
2. *After procedure ROUTE-REM applies rules (a)-(c) as many times as possible, the number of unrouted* 0_n *runs is less than or equal to the number of unrouted* 2_n *runs.*

Proof. The proof follows from Lemma 8 and the fact that each time we apply rules (a)–(c), the vertical density of the unrouted nets, the number of *runs*, and the number of *available* track slots decreases by one. □

Now procedure *ROUTE-REM* routes a subset of the unrouted 2_n *runs*, and all the unrouted 0_n *runs*. The procedure iterates as long as there are unrouted 0_n *runs*. At the beginning of each iteration G' is defined as the smallest rectangle inside R' such that all the terminal points located on the left and right boundary of R' from the neighbor nets in the unrouted 2_n *runs* can be horizontally projected to it; and all the terminal points from the nets in the unrouted 0_n *runs* can be vertically projected to it. Now project each of these terminal points to their nearest point in G'. A corner in G' is called a *doubly-covered* corner of G' if two of the previously identified terminal points were projected to that corner. Procedure *ROUTE-REM* routes all the nets in a 0_n and in a 2_n *run*. The specific *runs* routed depend on whether there is a *doubly-covered* corner in G'.

Case 1: There is a *doubly-covered* corner in G'.

Assume without loss of generality that the top-left corner of G' is a doubly-covered corner, since the other cases can be treated similarly. The 0_n and 2_n *run*

containing the terminal points projected to the top-left corner of G' are routed at this iteration. Figure 16(a) shows the case when no knock-knees are introduced in track p; Figure 16(b) shows the case when our procedure does not introduce a three-bend wire, but introduces a knock-knee in track p; and Fig. 16(c) shows the case when our procedure introduces a three-bend wire and a knock-knee in track p. The routing is in the track slots *matched* to the terminal points of the two neighbor nets in the 2_n *run*. The columns slots involved in the routing of these nets are those that are *matched* to the terminal points of the nets in the two *runs*. All of these slots are labeled *unavailable* at the end of the iteration.

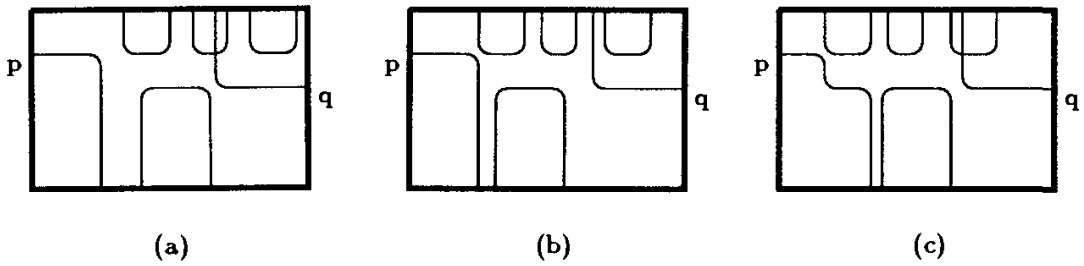


Fig. 16. Wiring for Case 1.

Case 2: There are no doubly-covered corners in G'.

The two cases are depicted in Fig. 17. The line segments intersecting the boundary of R indicate the terminal points from the previously identified nets projected to corners of G'. Assume without loss of generality that track t' is located above G'.

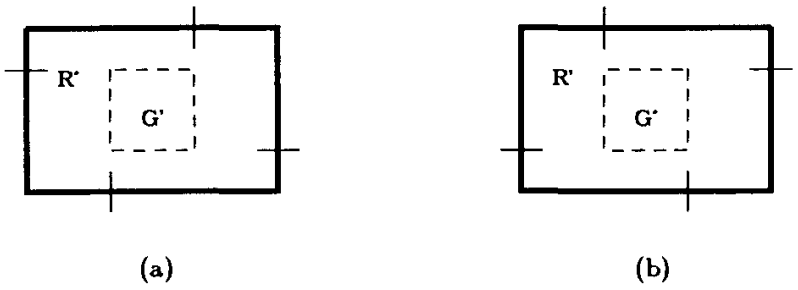


Fig. 17. There is no doubly-covered corner in G'.

If the case given in Fig. 17(a) ((b)) holds, we route the 2_n *run* that includes the net whose terminal point was projected to the top-left (top-right) corner of G', and the 0_n *run* that includes the net whose terminal point was projected to the top-right (top-left) corner of G'. In this case, track slot t' is used for routing, but after this step, label t' is assigned to track slot q. The specific routing is shown in Figs. 18(a)–(b). Here we only show the case when a neighbor net is connected by a three-bend wire.

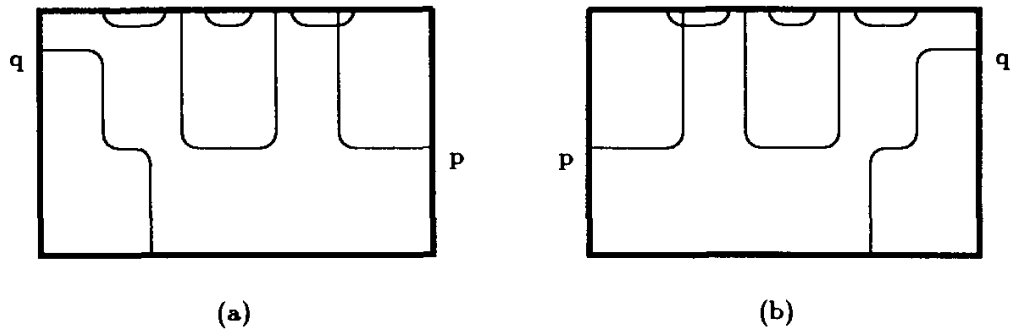


Fig. 18. Wiring for Case 2.

The above process is repeated until all the 0_n *runs* have been routed. In our example the *runs* 21-22-23, and 5 are routed as in Case 1. Figure 19 shows the wiring of the above two *runs*. If all the nets have been routed, then procedure *ROUTE-REM* and *ROUTE* terminate. Otherwise, each of the remaining 2_n *runs* is routed in the track slots *matched* to the neighbor nets in the *run*, and in the column slots *matched* to the nets in the *run*. All of these column and track slots are labeled *unavailable*. For our example, the remaining *run* 25-26-27 is routed using the track slots *matched* with the nets 25 and 27, the neighbor nets in the *run*. Figure 20 shows the wiring of the *run* 25-26-27. Procedures *ROUTE-REM* and *ROUTE* terminate

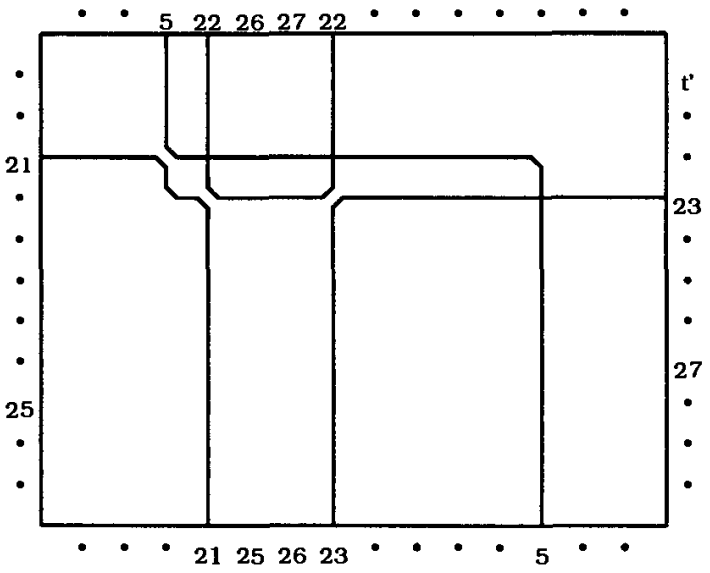


Fig. 19. Wiring of the runs after routing all 0_n runs.

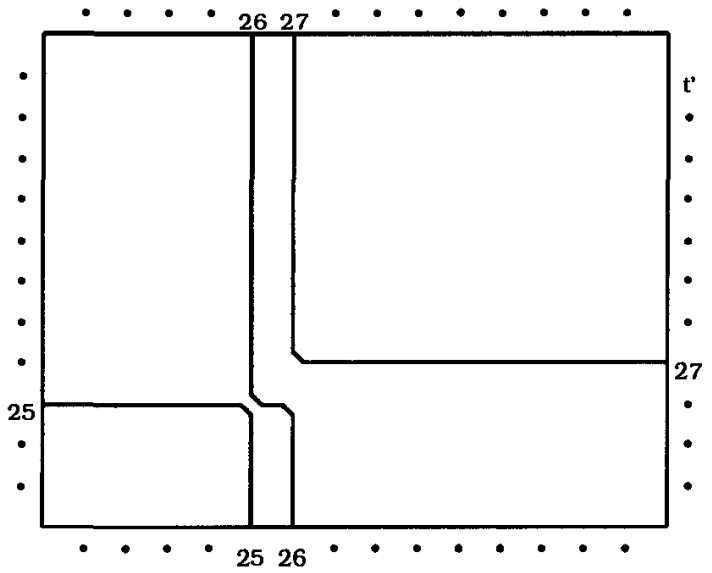


Fig. 20. Wiring of the remaining 2_n *runs*.

when all *runs* are routed. In Lemma 10 we establish that *ROUTE-REM* generates a 2-layout for I' and in Theorem 1, we establish that procedure *ROUTE* constructs a 2-layout for I.

Lemma 10. *Procedure ROUTE-REM routes problem instance I'.*

Proof. The proof follows from Lemmas 7–9 and the fact that the last two loops in *ROUTE-REM* route the remaining 0_n and 2_n *runs*. □

3.5. Summary

Theorem 1. *Procedure ROUTE constructs a 2-layout for any two-overlap routable instance $I = (R, N)$ of the SBR problem by adding at most two tracks and two columns in R. The time complexity of procedure ROUTE is $O(n)$, when the set of n terminal points is initially ordered.*

Proof. It is simple to show that all steps in procedure *ROUTE* can be implemented in $O(n)$ time, if the set of terminal points is initially ordered. The remaining part of the proof follows from Lemmas 2–6 and Lemma 10. □

Figures 21 and 22 illustrate the final wiring of the problems I' and I of our example.

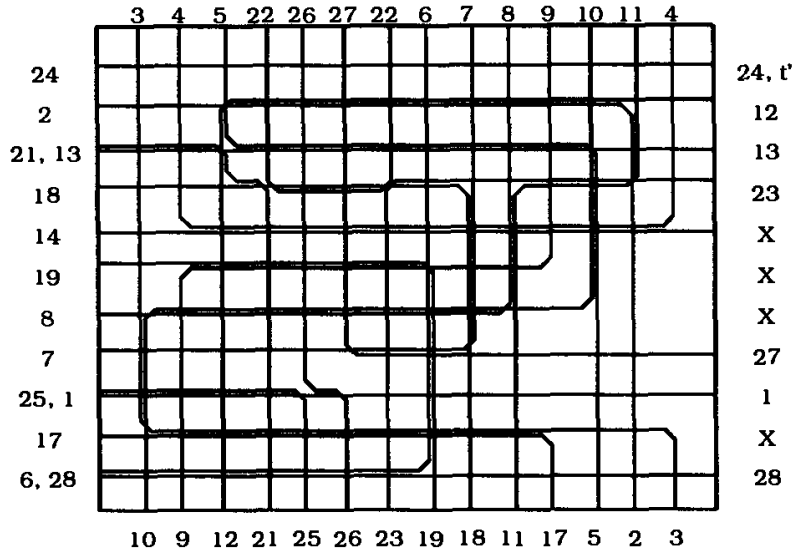


Fig. 21. Final wiring for the problem I'.

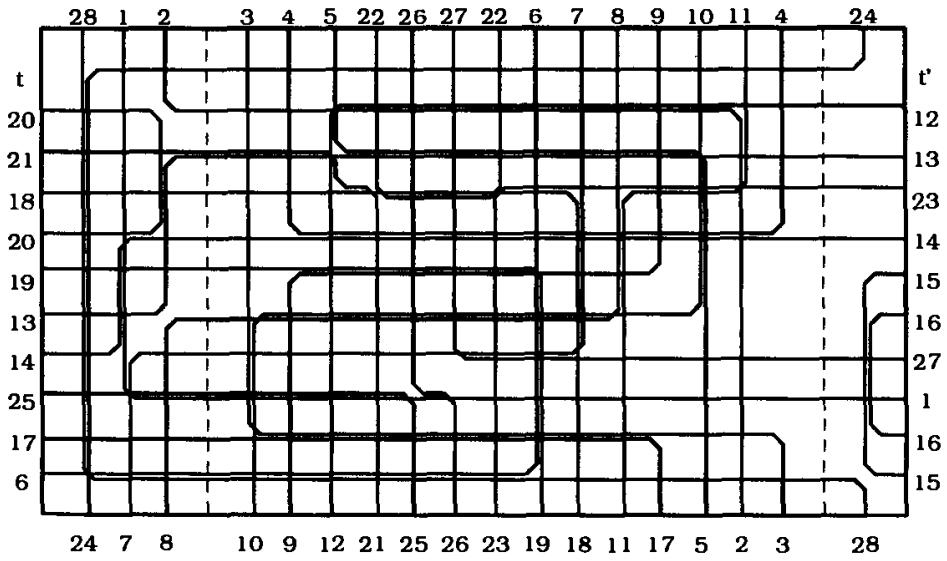


Fig. 22. Final wiring for the problem I.

4. Layer Assignment

The layer assignment phase comprises of assigning each wire segment to a layer in such a way that at each grid point no two wire segments and/or vias from different nets share a grid point in the same layer. The seven layers available for wiring are arranged from top to bottom in VHVHVHV order, where a layer is denoted V (H) if it has *vw*-segments (*hw*-segments) only. The three layers for the *hw*-segments are labeled a, b, and c; the layers for the *vw*-segments are labeled 1, 2, 3, and 4; and the layers are arranged from top to bottom, in the order 1, a, 2, b, 3, c, and 4. The

layer assignment of the wire segments introduced by procedures *ROUTE-LRTB* and *ROUTE-LR* inside R' is explained in Subsection (4.1). In Subsection (4.2) we explain our procedure for layer assignment of the wire segments introduced by procedure *ROUTE-REM* for the problem instance $I' = (R', N')$, and in Subsection (4.3) we explain the layer assignment for the wire segments introduced by procedures *ROUTE-LRTB* outside R'. For any particular problem instance, we perform the layer assignment of Subsections (4.1) and (4.3), or (4.2) and (4.3).

4.1. Layer Assignment for Wires Introduced by ROUTE-LRTB and ROUTE-LR Inside R'

Let us now consider the layer assignment of the wires, W', introduced by the procedures *ROUTE-LRTB* and *ROUTE-LR* inside R'. If the first (last) column has a *vw*-segment of an *LR* net which was introduced by *ROUTE-LRTB*, then we ignore that column when performing the layer assignment in this Subsection. Once the layer assignment of the remaining wires in R' is performed, the layer assignment for these columns is straightforward. This process greatly simplifies our notation. We begin by establishing in Lemma 11 some important properties of the wires introduced by procedures *ROUTE-LRTB* and *ROUTE-LR* inside R'. Then, we present our algorithm for the layer assignment.

Lemma 11. *The 2-layout constructed by procedures ROUTE-LRTB and ROUTE-LR inside R' satisfies the following properties.*

1. *Each wire consists of at most three segments. There are at most two hw-segments, and at most two vw-segments in each wire.*
2. *Each track (column) has hw-segments (vw-segments) from at most two nets.*

Proof. The proof is straightforward and is therefore omitted. □

Our procedure begins by assigning to three layers all the *hw*-segments, and then to four layers all the *vw*-segments. The *hw*-segments are assigned to layers by constructing a multigraph, and then coloring the edges of the multigraph. The *vw*-segments are assigned by considering each column at a time.

We construct a multigraph $G_{W'}$ from the 2-layout constructed by procedures *ROUTE-LRTB* and *ROUTE-LR* inside R'. Each track in the 2-layout is represented by a vertex in the multigraph $G_{W'}$. For each net with *hw*-segments assigned to the tracks represented by vertices v and w there is a distinct edge between vertices v and w. Note that there may be multiple edges between a pair of vertices in the graph. Since each net has at most two *hw*-segments in different tracks, and there are at most two *hw*-segments from two different nets in any track (Lemma 11), there can be at most two edges incident at any vertex in $G_{W'}$, and for each net there is at most one edge in $G_{W'}$.

A *coloration* of the edges of the multigraph is a function that assigns one of three colors (a, b, c) to each edge of the multigraph so that all edges incident on a vertex are colored differently. Since each node is of degree at most two, $G_{W'}$ can be colored in linear time (with respect to the number of nodes and edges) with the three colors. Each color corresponds to a horizontal layer, so the *hw*-segments of the net represented by an edge colored i is assigned to the layer i. The *hw*-segments of the remaining nets are assigned to horizontal layers in such a way that no two wire segments that overlap in a track are assigned to the same layer. By Lemma 11 we know that this is always possible.

The layer assignment for the *vw*-segments is performed by scanning the columns from right to left. In each column, the general rule for vertical layer assignment is that each *vw*-segment is assigned to a layer adjacent to the horizontal layer assigned to its adjacent *hw*-segment(s). Since all the *hw*-segments in a wire are assigned to one layer, the *vw*-segment can be assigned to one of two different layers. By Lemma 11, we know that each column has at most two *vw*-segments, and since there are two vertical layers adjacent to any horizontal layer, we know that this assignment is always possible. We formalize our results in Lemma 12.

Lemma 12. *Our procedure generates a seven-layer assignment for all the wire segments introduced by procedures ROUTE-LRTB and ROUTE-LR inside R' in $O(n)$ time.*

Proof. The proof follows from the above discussion and Lemma 11. □

4.2. Layer Assignment for Wires Introduced by ROUTE-REM Inside R'

Before discussing our layer assignment strategy for the wire segments introduced by *ROUTE-REM* in R', we define some useful terms, and prove properties of such 2-layout. In the 2-layout for problem instance I', constructed by the algorithm in Sec. 3, each neighbor net is either wired by an L shaped wire or by a monotone HVHV wire (i.e., a wire consisting of a horizontal segment attached to the left or right boundary, followed by a vertical segment, a horizontal segment and a vertical segment; and with the property that any vertical or horizontal non-grid line intersects the wire at most once). In the former case the net is called an *L-net*, and in the latter case it is called a *switching net (s-net)*. Note that our definitions of an L-net and an s-net are particular to a given 2-layout, i.e., the net which is an s-net in a 2-layout may not be so in another 2-layout. An s-net with a terminal point located on the left (right) boundary is called an *Ls-net (Rs-net)*. The track (column) where an s-net switches from one column (track) to the next is called an *s-track (s-column)*. The other track (column) where an Ls-net is routed is called the *beginning track or b-track* (*ending column or e-column*), and for an Rs-net it is called the *ending track or e-track* (*beginning column or b-column*). An Rs-net is said to be an *RBs-net (RTs-net)*, if it has a terminal located on the bottom (top)

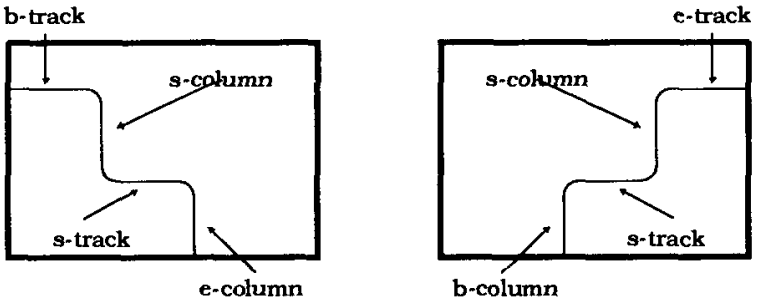


Fig. 23. *s*-net.

boundary. We define *LBs-nets and LTs-nets* similarly. Figure 23 illustrates the above definitions for an *s*-net.

An *L*-net with a terminal point located on the left (right) boundary of R' is called an *LL-net (RL-net).* The track where an *LL*-net (*RL*-net) is routed is called the *beginning track or b-track* (*ending track or e-track*) and the column where an *LL*-net (*RL*-net) is routed is called the *ending column or e-column* (*beginning column or b-column*). Remember that these definitions are particular to a given 2-layout. The leftmost (rightmost) column with a terminal point from a net in the set N'_{TB} is called the *beginning column or b-column* (*ending column or e-column*) of the net. Let us now establish some important properties for the 2-layouts constructed by *ROUTE-REM.*

Lemma 13. *The 2-layout constructed by procedure ROUTE-REM satisfies the following nine properties.*

1. *The s-column of an Ls-net (Rs-net) is the b-column (e-column) of a non-neighbor net uniquely associated with the s-net. The vertical wires of these two nets may overlap in at most one point in this column (i.e., they may form a knock-knee at that point).*
2. *Each column has vw-segments from at most two s-nets. In particular, if it is the s-column of two s-nets, then both are Rs-nets, or both are Ls-nets.*
3. *If a column is the s-column of two Rs-nets, then one is an RTs-net, and the other is an RBs-net. Furthermore, the e-track of the RBs-net is above the e-track of the RTs-net.*
4. *The s-track of an Ls-net (Rs-net) is the e-track (b-track) of an RL-net (LL-net) uniquely associated with the s-net. The horizontal wires of these two nets may overlap in at most one point in this track (i.e., they may form a knock-knee at that point).*
5. *Each track has hw-segments from at most two s-nets.*
6. *Ech grid point has at most one knock-knee.*
7. *Procedure ROUTE-REM introduces vw-segments in each column at most twice, and each time either one or two wire segments are added. When*

two segments are added in a single operation, then at most one of them belongs to an s-net.

8. *If a column has a trivial net, then the column does not have any other vw-segments.*
9. *The nets associated with two s-nets with the same s-column, do not overlap with each other in that column.*

Proof. Proof of (1): When procedure *ROUTE-REM* introduces an s-net, the switching takes place in order to accommodate the start of a net in the 0_n *run* at the s-column (see Fig. 16(c)), or to set up a new empty horizontal track that is necessary for the next step of the routing algorithm (see Fig. 18). Thus there is a net uniquely associated with the s-net that starts (ends) in the s-column of an Ls-net (Rs-net). Furthermore, by construction, we know that the two nets may overlap in at most one point in the s-column.

Proof of (2): Since the s-column of every s-net is the b-column or e-column of a non-neighbor net uniquely associated with the s-net (1), we associate the terminal point of the non-neighbor net located in the s-column with the s-net. By definition of an s-net, the b-column or e-column of an s-net has a terminal point from the s-net. Therefore, each of the two columns where an s-net is routed has a terminal point uniquely associated with the s-net. Since there are at most two terminal points in each column, it must be that there are vw-segments from at most two s-nets in a column.

When an Rs-net (Ls-net) is introduced by procedure *ROUTE-REM*, it is introduced on the right (left) side of the current rectangle G'. In order for an Rs-net and an Ls-net to have the same s-column, the column should be the right side of a rectangle G', and the left side of another rectangle G'. But, since each new rectangle G' is enclosed by the previous rectangle, and G' is never a single line segment, an Rs-net and an Ls-net cannot have the same s-column.

Proof of (3): From (1), we know that for each s-net there is a distinct net with a terminal point in the s-column of the s-net. Since there is at most one terminal point on the top and bottom ends of a column, the two Rs-nets with the same s-column must be an RTs-net and an RBs-net.

An RTs-net (RBs-net) is introduced by procedure *ROUTE-REM*, when the doubly covered corner is located on the bottom (top) right corner of the rectangle G'. Since each time an s-net is introduced, the new rectangle G' is enclosed within the previous rectangle G', it must be that the e-track of the RBs-net is above the e-track of the RTs-net.

Proof of (4): When procedure *ROUTE-REM* introduces an Ls-net (Rs-net), the s-track of the Ls-net (Rs-net) is the e-track (b-track) of an RL-net (LL-net) [see Figs. 16(c) and 18] uniquely associated with the Ls-net (Rs-net). Furthermore, the hw-segments of the two nets may overlap in at most one point in this track.

Proof of (5): Since the s-track of every s-net is the b-track or e-track of an L-net uniquely associated with the s-net (4), we associate the terminal point of

the L-net located in the s-track with the s-net. By the definition of an s-net, the b-track or e-track of an s-net has a terminal point from the s-net. Therefore, each of the two tracks where an s-net is routed has a terminal point uniquely associated with the s-net. Since there are at most two terminal points in each track, there are hw-segments from at most two s-nets in any track.

Proof of (6): Whenever procedure *ROUTE-REM* introduces a knock-knee at point p, at least one of the terminal points in that column belongs to the nets that form the knock-knee. Since each column has at most two terminal points, it must be that if p has two knock-knees, then the two terminals in that column belong to distinct nets, one from each knock-knee. Suppose that procedure *ROUTE-REM* introduces two knock-knees at grid point p. Let G'' be the rectangle G' in procedure *ROUTE-REM* when the first knock-knee was introduced at grid point p. If p is not a corner of G'', then we know that when the knock-knee was introduced at p, both the terminal points in the column of p belong to the nets in the knock-knee. Since every time a knock-knee is introduced by procedure *ROUTE-REM* at point p, at least one terminal point in column p belongs to the nets in the knock-knee, it cannot be that another knock-knee can be introduced at p. A contradiction, hence it must be that p was introduced at a corner c of G''. Assume that c is the top-left corner of G''. In this case, the terminal points "closest" to corner c (i.e., the one exactly above and exactly to the left of c) belong to the nets in the knock-knee. By arguments similar to the ones above, we know that the next knock-knee at p is introduced at a corner c' of a new rectangle G' (which we call new G''). Since the terminal point "closest" to corner c' belongs to one of the nets whose wires form the knock-knee, it must be that c' is the bottom-right corner of G''. Since at each iteration the new G' is within the previous G', we know that the new G'' must be a single point. But then, no knock-knee is introduced. A contradiction. Hence each grid point contains at most one knock-knee. This completes the proof of (6).

Proof of (7)–(9): The proof of (7) is straightforward. Every time procedure *ROUTE-REM* introduces a vw-segment, it also routes a net with a terminal point in that column. Therefore, if a column has a trivial net, then the column does not have any other vw-segments, and statement (8) holds. Statement (9) follows directly from (3). This completes the proof of (7)–(9) and the lemma. □

Our layer assignment procedure for the wire segments introduced by procedure *ROUTE-REM* for I' begins by assigning to three layers all the hw-segments and then to four layers all the vw-segments. One can easily assign all the hw-segments to three layers by considering each track at a time. However, this arbitrary assignment may not always lead to a feasible assignment of the vw-segments. In order to be able to generate quickly the vertical layer assignment, our horizontal layer assignment must satisfy some special properties. To perform the layer assignment of the hw-segments, we partition the set of s-nets in N' into four classes, such that the fourth class is a subset of the Rs-nets. Then the hw-segments of all the nets in N' are assigned to layers a, b, and c, in such a way that all hw-segments of each net are

assigned to the same layer, except for nets in the fourth class. The *hw*-segments of each fourth class *Rs*-net are assigned to layers *a* and *b*, or *c* and *b*. The *vw*-segments are assigned to layers 1, 2, 3, or 4 by scanning the columns from left to right. The general vertical layer assignment rule is as follows: each *vw*-segment is assigned to a layer adjacent to the horizontal layer(s) where its adjacent *hw*-segment(s) have been assigned.

4.2.1. Horizontal layer assignment

Before we present our horizontal layer assignment procedure *HLA*, we discuss our subprocedure that partitions the *s*-nets in N' into four classes, such that the fourth class contains only *Rs*-nets. This procedure begins by partitioning the *hw*-segments in each track into three groups; next, it constructs a multigraph, and then colors the edges in the multigraph with four colors. Let us now discuss these two steps in detail.

Since each grid edge contains at most two wire segments, and there is at most one knock-knee at each grid point (Lemma 13(6)), we know that all *hw*-segments in each track can be partitioned into three groups, such that the wire segments assigned to the same group do not overlap (not even at a single point). Furthermore, the partitioning of the segments is generated in linear time by scanning the track from left to right. Later on we may slightly modify this partition. Since an *s*-net has two *hw*-segments in different tracks, its *hw*-segments belong to groups in two different tracks. Obviously, the *hw*-segments of two *s*-nets assigned to a common track, may either belong to the same group or not.

Next, we construct a multigraph $G_{W'}$ from the 2-layout W' of I' generated by procedure *ROUTE-REM*. Each track in the 2-layout is represented by a vertex in the multigraph $G_{W'}$. For each *s*-net with *hw*-segments assigned to the tracks represented by vertices v and w there is a distinct edge between vertices v and w. Note that there may be multiple edges between a pair of vertices in the graph. Since each *s*-net has two *hw*-segments in different tracks, and there are at most two *hw*-segments from two different *s*-nets in any track (Lemma 13(5)), there can be at most two edges incident at any vertex in $G_{W'}$, and for each *s*-net there is exactly one edge in $G_{W'}$. The vertex corresponding to a track is labeled *d* if the *hw*-segments from two *s*-nets assigned to this track are in different groups (in the partitioning of the *hw*-segments in this track); otherwise, it is labeled *s*. Note that every vertex is labeled.

By Lemma 13(5) we know that the degree of each vertex is at most two. Therefore, the connected components of a multigraph $G_{W'}$ are either chains each with at least one vertex (Fig. 24(a)), or cycles (Fig. 24(b)). The tracks (*s*-nets) associated with a connected component of a multigraph are the tracks (*s*-nets) represented by the vertices (edges) in the component. The nets associated with a connected component are all of those nets with a *hw*-segment located on a track associated with the component.

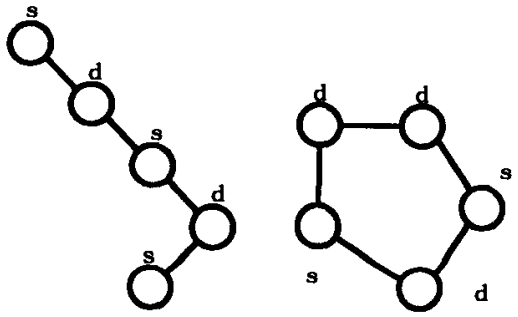


Fig. 24. Multigraph $G_{W'}$.

In order to wire our layout in seven layers we need to treat separately the components formed by a cycle with exactly one vertex labeled *d*. These cycles are called *single-d cycles*. What we would like to establish is that just before the first *Rs*-net in the single-*d* cycle was introduced by *ROUTE-REM*, the topmost and bottommost tracks represented by the vertices in the single-*d* cycle had at most one *hw*-segment from the *s*-nets in the cycle. We call this property the *ordering property*. In general one cannot prove that every single-*d* cycle satisfies the ordering property; however, in what follows we shall modify slightly the previously generated partitions of the horizontal tracks so that all the resulting single-*d* cycles in the multigraph satisfy the ordering property and the track partition is valid.

For each single-*d* cycle that does not satisfy the ordering property we perform the following operations. By definition the number of *s*-nets in the cycle must be at least two. If the first or second *s*-net routed by *ROUTE-REM* is an *Rs*-net, then it is simple to see that it satisfies the ordering property. So let us consider the remaining case, i.e., the first two *s*-nets in the single-*d* cycle routed by *ROUTE-REM* are *Ls*-nets. Let x and y be these two *s*-nets, and assume net x was routed by *ROUTE-REM* before net y. One of these nets is an *LBs*-net and the other is an *LTs*-net, as otherwise the component does not form a cycle. Suppose that the *s*-track of one of these two nets is the *b*-track of the other. If the *s*-columns of nets x and y are the same, then either the two *b*-tracks are labeled *d*, in which case the cycle is not a single-*d* cycle; or the horizontal wire segment in the *b*-track of one of these nets overlaps with at most one other *hw*-segment and the track is labeled *s*, in which case the wire can be assigned to a different group in the partition of that track and the new cycle has two vertices labeled *d*. If the *s*-columns of x and y are different, then the *s*-column of x is to the left of the *s*-column of y. It is simple to verify that the *hw*-segment of net x located in its *b*-track overlaps with at most another horizontal segment and can be reassigned in the partitioning of the segments in that track so that the new cycle has either zero or two vertices labeled *d*. In some cases the segment just to the right of the segment reassigned may also need to be reassigned. So we need to consider only the case when the *s*-tracks of these *s*-nets (x and y) is not the same as the *b*-track of the other, and one of these

s-nets is an *LBs*-net and the other is an *LTs*-net. Since the two *b*-tracks have only one wire from the *s*-nets in the cycle, there must be at least three *s*-nets in the cycle. If the next *s*-net from the simple-*d* cycle routed by $ROUTE - REM$ is an *Ls*-net, then its *b*-track must be different than the *b*-tracks of x and y. Since the remaining *s*-nets in the cycle will only be introduced inside the G' at this point, it then follows that the component does not form a cycle. So this third net must be an *Rs*-net, and by the above discussion the simple-*d* cycle satisfies the ordering property. Therefore, after performing the above transformations all the single-*d* cycles in the resulting multigraph satisfy the ordering property.

A *coloration* of the edges of the multigraph is a function that assigns one of four colors to each edge of the multigraph so that each edge is labeled with one of the first three colors, and for each degree-two vertex labeled *s* (*d*) the two edges incident to it are colored identically (differently). The only exceptions to this rule are the single-*d* cycles, in which case one of the edges representing an *Rs*-net is labeled with the fourth color and the vertices labeled *s* adjacent to this edge do not need to satisfy the property that their edges are colored with the same color.

Let us now outline our procedure to color the edges of the multigraph with four colors. First, each chain is transformed into a cycle by adding two (dummy) vertices labeled *d* adjacent to each other, and each one adjacent to one of the end vertices in the chain. After this operation all the components are cycles and each cycle has at least one vertex labeled *d*. Each non single-*d* cycle is transformed as follows. Each path from v to w with all vertices labeled *s* except for v and w is replaced by a single edge adjacent to v and w (all nodes labeled *s* in the path are deleted). As a result of this, all these cycles have at least two vertices and all their vertices are labeled *d*. It is simple to show that these cycles can be easily colored with the first three colors. The edges in the original components corresponding to these cycles are colored with the same color as the edge that replaced them. The dummy vertices and edges are then deleted. The cycles with exactly one vertex labeled *d* are colored differently. First we find the *Rs*-net with the rightmost *s*-column amongst all the *Rs*-nets represented by the edges in the cycle. The edge corresponding to this *Rs*-net is assigned the fourth color. Relabel the vertices adjacent to this edge *d* and color the other edges in the cycle by the procedure used for the case when the cycle is not a single-*d* cycle.

We classify the *s*-nets by assigning all *s*-nets corresponding to edges colored i to the i^{th} class. Procedure *HLA* for the layer assignment of the *hw*-segments based on this classification of the *s*-nets is defined below.

procedure HLA

Partition the *hw*-segments in each track into three groups;
Construct the multigraph $G_{W'}$;
Modify slightly the partition as described earlier so that all the remaining single-*d* cycles in the resulting multigraph $G_{W'}$ satisfy the ordering property;
Color the edges of the multigraph edges using the procedure discussed above.

Assign the s-nets represented by edges colored i to the i^{th} class.
while there are unassigned hw-segments of fourth class s-nets **do**
$k \leftarrow$ rightmost s-column of a fourth class s-net with an unassigned hw-segment.
/* In Lemma 14(1) we prove that all the hw-segments of fourth class nets whose s-column is column k have not been assigned. */
Let n'_i be a fourth class s-net with its s-column in column k;
Let C be the component of n'_i.
Let $n'_{i'}$ be the net associated with n'_i. /* the net identified in Lemma 13(1) */
/* Net n'_i is represented by an edge in $G_{W'}$ that is part of a single-d cycle. */
/* In what follows we assign to layers the hw-segments of a subset of nets that includes n'_i and possibly $n'_{i'}$, depending on the vw-segments in column k. */
There are two cases depending on the vw-segments in column k.
Case 1: The only vw-segments in column k belong to nets n_i' and $n'_{i'}$.
Assign the hw-segments of n'_i in the s-track (e-track) to layer a (b).
Map the first three colors to the layers consistent with the assignment of n'_i.
Assign to layers the hw-segments of the remaining s-nets associated with component C. The assignment must be consistent with the above mapping.
For each track t associated with component C, assign to layers the remaining hw-segments. The assignment must be consistent with the assignment defined so far and the previously constructed grouping (first step) for track t.
/* Note that the hw-segment of net $n'_{i'}$ might not be assigned. */
Case 2: There are three or four vw-segments in column k.
Let n'_j be the other s-net with vw-segments in column k, unless there are no other s-nets with this property in which case n'_j is the net other than n'_i and $n'_{i'}$ with vw-segment in column k.
If the hw-segment(s) of n'_j has not been assigned, and if n'_j is not an s-net whose vw-segment overlaps (in more than one point) with that of $n'_{i'}$ in column k,
then Assign the hw-segment of net n'_j to layer b, unless n'_j is a fourth class s-net in which case assign the hw-segments of n'_j in the e-track and s-track to layers b and c, respectively.
Let C' be the component where n'_j belongs.
Assign all the hw-segments in all tracks in C' as in Case 1.
If the hw-segments of n'_j have not been assigned, or have been assigned to either assigned to either layer b or c, **then** execute the steps of Case 1.
Otherwise, execute the steps of Case 1, but the hw-segments of n'_i are assigned to layers c and b rather than to layers a and b.
end while
for each component C with an unassigned hw-segments in one of its tracks **do**
/* In Lemma 14(2) we show that all the hw-segments of the nets associated with component C are not assigned. */
Let M be any mapping of the first three colors to the horizontal layers.
Proceed as in Case 1 to assign to layers all the hw-segments of the nets in C.
end for
end of procedure HLA

Figure 25 illustrates the layer assignment of the *hw*-segments of the wires in the routing layout for I' given in Fig. 21. Lemma 14 establishes the correctness of procedure HLA, and Lemma 15 establishes some important properties which we use to show that the vertical layer assignment is feasible.

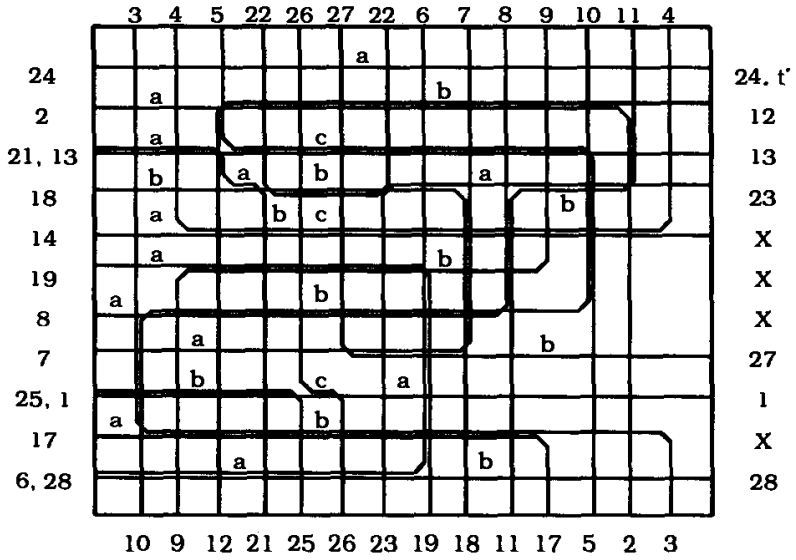


Fig. 25. Layer assignment of all *hw*-segments.

Lemma 14.

(a) At the beginning of each iteration of the while-loop and the for-loop of procedure HLA, and at the end of procedure HLA, the following statements hold.

1. *There is a vertical line l that partitions R' into two rectangles such that all the hw-segments of the fourth class s-nets in the right (left) rectangle have been assigned (not been assigned) to layers. Furthermore, at the beginning of the i^{th} iteration of the while-loop, the right rectangle of R' defined by l includes only the rightmost $i-1$ s-columns of fourth class nets.*
2. *Either all or none of the hw-segments of nets associated with a component have been assigned to layers.*

(b) At the end of the while-loop of procedure HLA, (1) holds with the right rectangle being the entire rectangle R'.

(c) At the end of the procedure HLA, all the hw-segments of W' have been assigned to layers.

Proof. Clearly (a) holds at the beginning of the while-loop. Let us now show that if (a) holds at the beginning of the while-loop, then it holds at the end of the loop. At each iteration we assign all the *hw*-segments in the tracks associated with

the components that include the fourth class nets with their s-column in column k and possibly those in another component. All the components whose tracks are assigned in this iteration include at least one net with a vw-segment in column k. By construction each component has at most one fourth class net. Therefore, to complete the proof we need to show that the components assigned in this iteration either include a fourth class net with its s-column in column k, or do not include a fourth class net. Suppose this is false. Suppose that component C assigned in this iteration has a fourth class net with its s-column to the left of column k. This component must include net n'_j which is not an s-net. Net n'_j was introduced by $ROURE - REM$ when some s-net x in C was routed. Since the vw-segments of nets n'_j and n'_i, overlap in column k and both of this nets have a terminal point at that column, it must be that the track with a terminal point from net x is above (below) the e track of n'_i when x has a terminal point on the bottom (top) side of R'. It cannot be that x is an Rs-net, as otherwise its s-column would be to the right of column k. So x is an Ls-net. Since the cycle satisfies the ordering property it must be that when the first Rs-net in the cycle is routed by *ROUTE-REM*, there is only one wire segment in the b-track of net x from the s-nets in C. Since C forms a cycle, such a track must contain two hw-segments from the s-nets in the cycle. So this Rs-net is introduced before n'_i, or the component is not a cycle. A contradiction. Therefore (a) holds at the end of the first loop of procedure HLA. Since (a) holds at each iteration, (b) holds. The proof for (c) is straight forward and thus omitted. □

4.2.2. Vertical layer assignment

First, in Lemma 15, we establish some properties of the horizontal layer assignment generated by procedure HLA, which are used in the vertical layer assignment procedure.

Lemma 15. *The following statements hold after procedure HLA.*

1. *Each column has wires from at most two s-nets that belong to the fourth class.*
2. *The hw-segments of each fourth class net are assigned to layers a and b, or c and b.*
3. *Any three hw-segments of three distinct nets adjacent to three vw-segments that overlap at a grid point, have not been assigned to the same layer.*

Proof. By Lemma 13(2) and the fact that each net represented by a vertex colored with the fourth color is an Rs-net, we know that there can be no more than two s-nets in any column. Therefore, statement (1) holds. From procedure HLA, we know that the hw-segments of each net from the fourth class are assigned to layers a and b (or c and b). Therefore, statement (2) holds. When three vw-segments overlap at a grid point, two of these vw-segments belong to two nets that

form a knock-knee at that grid point. By procedure HLA, the two hw-segments that form the knock-knee are assigned to different layers. If both nets that form the knock-knee do not belong to the fourth class, then clearly statement (3) holds. Otherwise, at most one of them is a fourth class s-net. From Case 2 of procedure HLA, we know that three hw-segments of these three nets have not been assigned to the same layer. Therefore, statement (3) holds. □

Next, we explain the layer assignment of the vw-segments (procedure VLA). As mentioned before, the vw-segments are assigned to one of four layers, and the layer assignment is performed by scanning the columns from left to right. In each column, the general rule for vertical layer assignment is: each vw-segment is assigned to a layer adjacent to each horizontal layer assigned to its adjacent hw-segment(s). For example, if the hw-segments of a net are assigned to layers a and b, then the vw-segment adjacent to both of these segments is assigned to layer 2. On the other hand, if the hw-segment(s) is (are) assigned to layer a, then the vw-segment can be assigned to layer 1 or layer 2. In what follows, we show that there is a feasible layer assignment consistent with this general rule, and that such assignment can be generated quickly. A feasible assignment is one that satisfies the multilayer wiring conditions given in Sec. 2. There are three cases depending on the number of vw-segments in the column.

Case 1: There are at most two vw-segments in the column.

Each vw-segment is assigned to a layer by following the general rule together with the restriction that both segments are assigned to different layers. Since there are at most two vw-segments of two nets, at most one belongs to an s-net (Lemma 13(7)). Therefore, a feasible assignment exists.

Case 2: There are three vw-segments in the column.

Each vw-segment is assigned to a layer by following the general rule together with the restriction that no two segments that overlap are assigned to the same layer. By Lemma 15(3), we know that no three hw-segments of distinct nets that are adjacent to vw-segments that overlap at a grid point are assigned to the same layer. If one of the vw-segments belongs to a fourth class net, then it is assigned to layer 2 (3) if the hw-segments of the fourth class net are assigned to layers a and b (c and b). One can easily show that the remaining vw-segments can be assigned to layers as per the general rule. On the other hand, it is simple to show that a feasible assignment exists if not all the hw-segments adjacent to the vw-segments in this column are assigned to the same layer. Otherwise, we know that no two of them are part of a knock-knee in this column. Assume without loss of generality that all the hw-segments adjacent to the vw-segments are assigned to a. Since there is at most a two-overlap, it must be that at least two of the three vw-segments do not overlap. These wire segments are assigned to layer 1, and the third is assigned to layer 2. Therefore, a feasible assignment consistent with our rules exists. Figure 26 shows the layer assignment of the leftmost three columns of the 2-layout of Fig. 21,

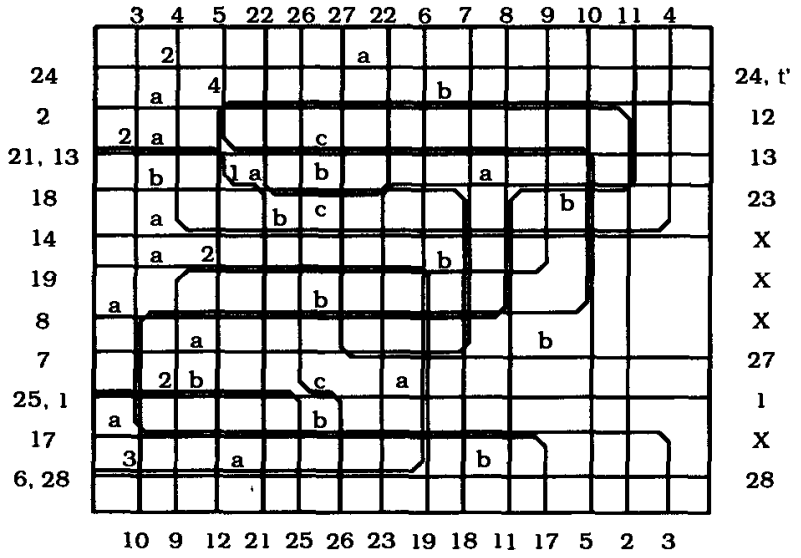


Fig. 26. Layer assignment for wire segments of three columns.

the first two of which are assigned as described in Case 1, and the third column is assigned as described in Case 2.

Case 3: There are four *vw*-segments in the column.

Each *vw*-segment is assigned to a layer by following the general rule together with the restriction that no two segments that overlap are assigned to the same layer. By Lemma 15(3), we know that no three *hw*-segments of distinct nets that are adjacent to *vw*-segments that overlap at a grid point are assigned to the same layer. This implies that all *vw*-segments in this column that are adjacent to *hw*-segments assigned to the same layer can be partitioned into two groups, each of which can be assigned to one of the vertical layers adjacent to the horizontal layer. Hence we can assign the *vw*-segments to layers adjacent to the horizontal layers where the *hw*-segments adjacent to these *vw*-segments are assigned, i.e., the vertical layer assignment is consistent with the general vertical layer assignment rule.

By Lemma 13(7), these three cases include all possibilities of the number of *vw*-segments in a non-empty column. The above process is repeated until all non-empty columns have been considered, at which point procedure *VLA* terminates.

From the above procedures we know that all the *vw*-segments (*hw*-segments) were assigned to the layers 1, 2, 3, and 4 (*a*, *b*, and *c*). Figure 27 illustrates the layer assignment of all the nets for the layout for problem I' given in Fig. 21.

Lemma 16. *Procedures HLA and VLA generate a feasible seven-layer assignment for all the wire segments in* R' *in* $O(n)$ *time.*

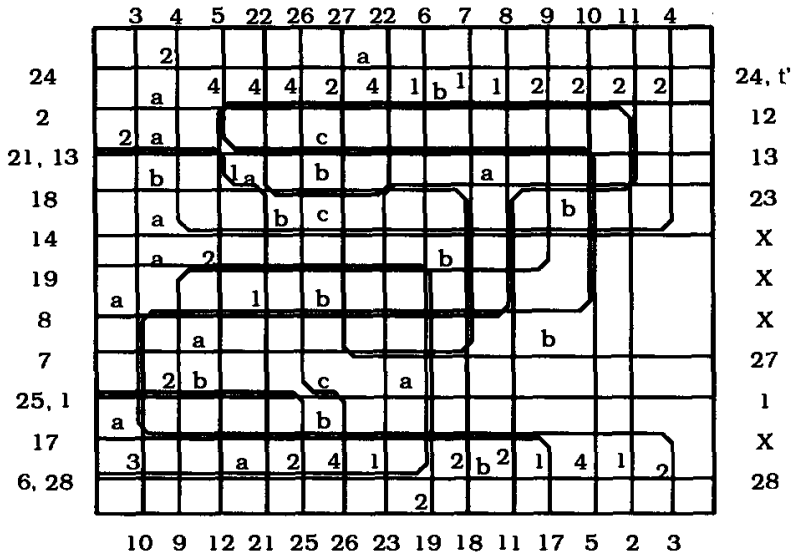


Fig. 27. Layer assignment of problem I'.

Proof. It is simple to show that all the steps in procedures HLA and VLA can be implemented to take $O(n)$ time. The remaining part of the proof follows from Lemmas 13, 14, 15, and the discussion before this lemma. □

4.3. Layer Assignment for Wires Introduced by ROUTE-LRTB Outside R'

Let us now consider the layer assignment for the wire segments introduced by *ROUTE-LRTB* outside R'. We consider only the wire segments located to the left of the leftmost vertical stretching line, since the wire segments to the right of the rightmost stretching line can be handled similarly. We shall refer to this rectangle as R''. Before we present our layer assignment strategy, we define some terms, and prove some properties of the 2-layouts generated by *ROUTE-LRTB* in R''. It is important to note that the additional column is needed in order for the *hw*-segments to change layers. If this column is not introduced, then it is impossible to perform the layering without making major changes to the layer assignment of R'. To convince yourself of this fact try to wire the column with nets 1 and 7 in Fig. 28 without allowing a change of layers in the layer assignment of R'.

Procedure *ROUTE-LRTB* introduces wires in R'' with at most three wire segments, of which at most two are *hw*-segments and at most two are vertical (Lemma 17). A net is called an *s*-net if it contains three wire segments in R'', and two of them are horizontal. Note that our definition of an *s*-net is particular to a given 2-layout, i.e., the net which is an *s*-net in a 2-layout may not be so in another 2-layout. When *ROUTE-LRTB* in R'' routes an *s*-net, it is because the *s*-net includes a terminal point on the left boundary of R'', and which is located on the track corresponding to the upper or lower boundaries of rectangle G. The tracks where an *s*-net has its *hw*-segments are called the *b*-track and the *e*-track. The *b*-track is

the one that includes the above terminal point. The s-column of an s-net is the column where the s-net has its vw-segment. The corner of the wire for an s-net in its b-track is marked upper (lower) if the b-track is the track corresponding to the upper (lower) boundary of G. A t-net associated with an s-net is a net connected by a vertical-horizontal wire or a vertical-horizontal-vertical wire, whose leftmost terminal point is in the s-column of the s-net, and whose hw-segments are not in any of the interior tracks of the rectangle G defined when the s-net was introduced.

Our layer assignment procedure begins by assigning the hw-segments intersecting the stretched column to layers a or c. This assignment should be consistent with the layer assignment generated in Subsections (4.1) and (4.2). We shall refer to this assignment as a tentative assignment. This assignment may change (i.e. a could become b or c could become b, but not both) and eventually becomes the final assignment. Then, the procedure performs the layer assignment of all the remaining wire segments while scanning the columns in R'' from right to left. At each column, we define an upper track and a lower track. At the rightmost column of the rectangle R'', the upper (lower) track is one of the interior tracks of the last rectangle G defined in procedure *ROUTE-LRTB*. In case the last G has only two tracks, then the upper and lower tracks are a hypothetical track between the two tracks of the rectangle G. At column k, the upper track is the topmost track where the marked upper corner of an s-net with s-column k is located, unless there are no such nets in which case the upper track is the same as the upper track of column $k+1$ (i.e., the previous column). The lower track is defined similarly. All the hw-segments with a point to the right of column k, and between (inclusive) the upper and lower tracks of column $k+1$ have the final assignment. The remaining hw-segments have a temporary assignment. Then, at each iteration we make a final assignment of all the hw-segments in the tracks between (inclusive) the upper and lower tracks of the current column, with a point to the right of the current column. All the hw-segments assigned to layers in the temporary assignment are assigned to layers a or c. One of these assignments may change to b. Therefore, whenever we make a temporary assignment we should make sure that it is possible to change it to b. Before we present our procedure, we establish some properties of the 2-layout generated by *ROUTE-LRTB* in R''.

Lemma 17. *The 2-layout constructed by procedure ROUTE-LRTB in R'' satisfies the following properties.*

1. *Each wire consists of at most three segments. There are at most two hw-segments and at most two vw-segments in each wire.*
2. *Each wire with two horizontal segments is an s-net and it has a terminal point on the left boundary of rectangle R''. In the s-column of an s-net there could be a vw-segment of a t-net. Either these two nets form a knock-knee or their vw-segments do not overlap.*
3. *At most two s-nets have the same s-column.*

4. *There are at most two sets of nets with vw-segments in a column. Each set contains at most two nets, and if there are two nets, then one is an s-net and the other is a t-net.*
5. *If there are two s-nets at a column then the vw-segments of one of the s-nets may overlap (at more than one point) with the vw-segments of the t-net associated with the other s-net; however, both nets do not satisfy this property.*

Proof. The proof is straightforward and is therefore omitted. □

It is simple to show that if there is an s-net with its corner point marked upper (lower) at a column, then the b-track of the s-net is above (below) the upper (lower) track in the column immediately to its right. Let us now discuss our procedure to perform the layer assignment when considering column k. There are three cases depending on the number of s-nets with vw-segments at column k.

Case 1: There are no s-nets with s-column at column k.

By Lemma 17(4) and the conditions of the case, we know that there are vw-segments from at most two nets at column k and neither of the nets is an s-net. Therefore, the upper (lower) track at column k is equal to the one at column $k+1$ (i.e. the previous column). Let x and y be the two nets with vw-segments in column k. In case there is only one net, ignore the operations for net y. If the hw-segments of net x (y) have not been assigned, then assign them to layer a (c) and their vw-segment in column k to layer 2 (3). Note that this allows us to change them to b later on, if they are still temporarily assigned. Assign the remaining vw-segments to layers consistent with the previous assignments. This is always possible since there are two vertical layers adjacent to every horizontal layer.

Case 2: There is only one s-net with s-column at column k.

By Lemma 17(4) we know that there are vw-segments from at most three nets. By the conditions of the case, we know that there is an s-net and its corresponding t-net, and another net (x). Assign the hw-segment of the s-net in the b-track to the same layer as the hw-segment in its e-track, unless it has not been assigned, in which case both are assigned to layer a or layer c. As a result of this assignment, a temporary assignment of hw-segments located in the b-track of the s-net may have to be changed to layer b. Note that after this assignment, that track will have its final assignment. If the hw-segment of net x has not been assigned, then assign it to layer a. Assign its vw-segment to either layer 2 or 3, consistent with the assignment of its hw-segment. Now assign the vw-segments of the s and t nets. This assignment is always possible, since all the hw-segments in each of these two nets have been assigned to the same layer and there are two vertical layers adjacent to every horizontal layer.

Case 3: There are two s-nets with their s-column at column k.

By Lemma 17(4), we know that there are vw-segments from at most four nets, two pairs of s-t nets. Perform the horizontal layer assignment for each pair, as

in Case 2. By Lemma 17(5), we know that there is at most one s-net whose vw-segment overlaps at more than one point with that of the t-net in the other pair. Let s_1 be that s-net, unless no such net exists, in which case s_1 is either of the two s-nets. Assign the vw-segment of the net s_1 to layer 2 or 3, depending on the layer where its hw-segment have been assigned. The assignment of the other vw-segments proceeds as in Case 2.

As mentioned before, the above process is repeated on the other side of rectangle R'. Figure 28 illustrates how our procedure handles the layer assignment of the first three columns of R. Cases 3, 2, and 1 arise when considering the third, second, and first columns of R, respectively. Lemma 18 establishes the main result in this Section.

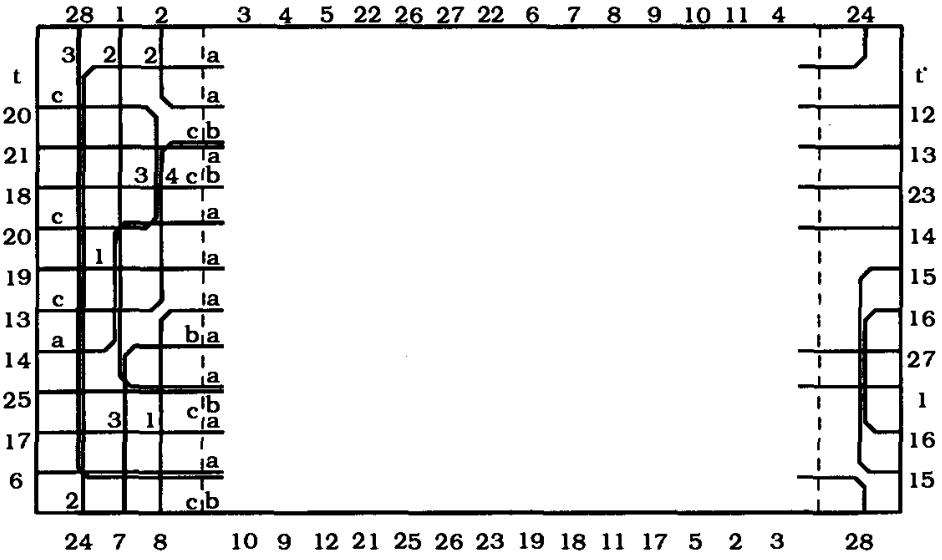


Fig. 28. Layer Assignment of wires introduced by *ROUTE-LRTB*.

Lemma 18. *Our procedure generates a feasible seven-layer assignment for all the wire segments introduced by ROUTE-LRTB inside R'' in $O(n)$ time.*

Proof. The proof follows from the above discussion and Lemma 17. □

4.4. Complete Layer Assignment

For any particular problem instance, we perform the layer assignment of Subsections (4.1) and (4.3), or (4.2) and (4.3). Figure 29 shows the complete layer assignment of all the nets for the routing layout of problem I given in Fig. 22.

Theorem 2 establishes our result in this Section, and Theorem 3 establishes the main result of this paper.

Theorem 2. *Our procedure constructs a seven-layer wiring for any 2-layout generated by procedure ROUTE. The time complexity of our algorithm is linear*

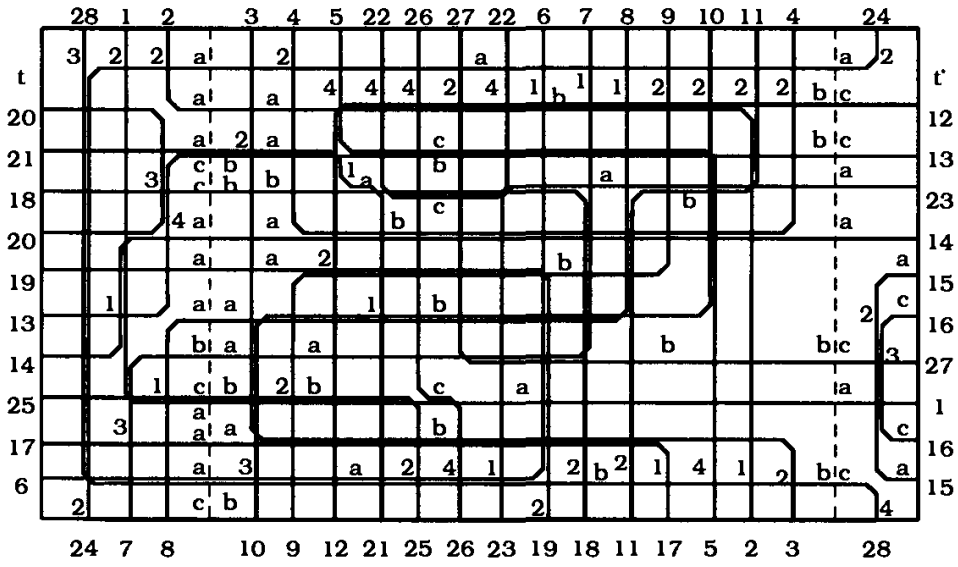


Fig. 29. Layer assignment.

with respect to n, the number of terminal points (if the set of terminal points is initially ordered). The 2-layout contains no more than $1.5v^*$ *bends where* v^* *is the natural bend number for I.*

Proof. Procedure *ROUTE-REM* is the only procedure that connects nets with wires with a number of bends that exceeds their natural bend number. By Lemma 13(1) and (4), and procedure *ROUTE-REM*, we know that for every net connected by a wire whose number of bends exceeds the natural bend number of the net, the procedure connects at least two nets by wires with a number of bends equal to the natural bend number of the nets. One of these wires has one bend and the other has two bends. Therefore, $1.5v^*$ is an upper bound on the number of bends introduced by the algorithm. The proof of the time complexity bound is straightforward. The remaining part of the proof follows from Lemmas 12, 16, and 18. □

Theorem 3. *A seven-layer wiring for any two-terminal-net two-overlap routable SBR problem can be constructed in* $O(n)$ *time, if the set of n terminals is initially ordered. The layout has at most two extra tracks and two additional columns.*

Proof. By Theorems 1 and 2. □

5. Discussions

We presented an algorithm that given any two-overlap routable *SBR* problem instance, adds at most two tracks and two columns, and wires it in seven layers. The time complexity for our algorithm is $O(n)$ when the set of n terminal points is

initially ordered. The constant associated with the time complexity bound is small. All the wires in each layer are either vertical or horizontal, but not both. Most of the vias introduced by our algorithm join wires in adjacent layers. The rest of the vias join wires in adjacent horizontal layers, and are located along two columns. It may be possible to perform the wiring in seven layers by adding only one additional track or column slot; however, such a procedure is extremely complex.

Our algorithm can be generalized to solve the multiterminal-net *SBR* problem under the two-overlap, by splitting the multiterminal nets into two-terminal nets and stretching the layout,[20,21] multiterminal nets can be partitioned into two-terminal nets. By stretching the grid of R, our algorithm can be directly applied to obtain seven-layer wiring solutions. Our algorithm can be easily adapted to solve the two-stacked pin two-overlap *SBR* problem when the set of terminals is compatible. In this case, the set of terminals is said to be compatible if for each track or column only one of the endpoints has terminal points. Any multiterminal *SBR* problem with a set of compatible terminals can be transformed to a two-stacked pin *SBR* problem by duplicating midpoints of a multiterminal net. Our algorithm can be modified to handle this problem. We have also generalized our algorithm to handle higher number of overlappings. For brevity, we do not discuss further generalizations of our algorithm.

References

1. M. Sarrafzadeh, "Channel-routing problem in the knock-knee mode is NP-Complete", *IEEE Transactions on Computer Aided Design* **CAD-6(4)** (1987) 293–303.
2. T. G. Szymanski, "Dogleg channel routing is NP-complete", *IEEE Transactions on Computer-Aided Design* **4(1)** (1985) 31–41.
3. B. S. Baker, S. N. Bhatt and F. T. Leighton, "An approximation algorithm for Manhattan routing", *Advances in Computer Research*, Ed. F. P. Preparata, **2** (1984) 205–229.
4. M. Burstein and P. Pelavin, "Hierarchical wire routing", *IEEE Transactions on Computer-Aided Design* **CAD-2(4)** (1983) 223–234.
5. D. N. Deutsch, "A Dogleg channel router", Proceedings of the 13th Design Automation Conference (1976) 425–433.
6. R. L. Rivest, A. E. Baratz and G. Miller, " Provably good channel routing algorithms", in *VLSI Systems and Computations*, ed. H. T. Kung, R. Sproull and G. Steele, Computer Science Press, Rockville, Maryland (1981).
7. J. Soukup and J. C. Royle, "On hierarchical routing", *J. Digital Systems* **5(3)** (1981).
8. T. Yoshimura and E. Kuh, "Efficient algorithms for channel routing", *IEEE Transactions on Computer-Aided Design* **CAD-1(1)** (1982) 25–35.
9. R. L. Rivest and C. M. Fiduccia, "A greedy channel router", *Proceedings of the 19th Design Automation Conference* (1982).
10. W. K. Luk, "A greedy switch-box router", *INTEGRATION, the VLSI journal* **3** (1985).
11. A. Hashimoto and J. Stevens, "Wiring routing by optimizing channel assignment within large apertures", Proceedings of the 8th Design Automaton Conference (1971) 155–160.

12. T. F. Gonzalez and S. Q. Zheng, "A near-optimal algorithm for routing compatible nets inside a rectangle", UCSB Technical Report #TRCS 87-14 (1987).
13. A. Frank, "Disjoint paths in rectilinear grid", *Combinatorica* **2(4)** (1982) 361–371.
14. F. P. Preparata and W. Lipski, Jr, "Optimal three-layer channel routing", *IEEE Transaction on Computers* **33(5)** (1984).
15. K. Mehlhorn, F. P. Preparata and M. Sarrafzadeh, "Channel routing in knock-knee mode: Simplified algorithms and proofs", *Algorithmica* **2(1)** (1986) 213–221.
16. T. F. Gonzalez and S. Q. Zheng, "Simple three-layer channel routing algorithms", *Proceedings of the 3rd AEGEAN Workshop on Computing* (*AWOC 88*), Lecture Notes in VLSI Algorithms and Architectures, Springer-Verlang (1988) 237–246.
17. R. Kuchem, D. Wagner and F. Wagner, "Area optimal three layer channel routing", *Proceedings of the Symposium on Foundations of Computer Science* (1989) 506–511.
18. S. Gao and M. Kaufmann, "Channel routing of multiterminal nets", *Proceedings of the 28th Symposium on Foundations of Computer Science* (1987) 316–325.
19. C. Wieners-Lummer, "Three-layer channel routing in knock-knee mode", Technical Report, University of Paderborn, Germany.
20. K. Mehlhorn and F. Preparata, "Routing through a rectangle", *Journal of ACM* **33(1)** (1986) 60–85.
21. T. F. Gonzalez and S. Q. Zheng, "Grid stretching algorithms for routing multiterminal nets through a rectangle", *INTEGRATION: the VLSI Journal* **13(2)** (1992).
22. M. L. Brady and D. J. Brown, "VLSI routing: Four layers suffice", *Advances in Computing Research*, Ed. F. P. Preparata **2** (1984).
23. M. L. Brady and D. J. Brown, "Optimal multilayer channel routing with overlap", in *Advanced Research in VLSI*, Ed. C. E. Leiserson, The MIT Press (1986).
24. .D. Braun, J. Burns, S. Devadas, H. K. Ma, K. Mayaram, F. Romeo and A. Sangiovanni-Vincentelli, "A new multi-layer channel router", *Proceedings of the 23rd Design Automation Conference* (1986).
25. P. Bruell and P. Sun, "A greedy three layer channel router", *Proceedings of the International Conference on Computer-Aided Design* (1985).
26. Y. K. Chen and M. L. Liu, "Three-layer channel routing", *IEEE Transactions on Computer-Aided Design* **3 (2)** (1984) 156–163.
27. J. Cong, D. F. Wong and C. L. Liu, "A new approach to three and four-layer channel routing", *IEEE Transactions on Computer-Aided Design* **7(10)** (1988) 1094–1104.
28. R. J. Enbody and H. C. Du, "Near optimal n-layer channel routing", *Proceedings of the 23rd Design Automation Conference* (1986) 708–714.
29. S. E. Hambrusch, "Channel routing algorithms for overlap models", *IEEE Transactions on Computer-Aided Design* **CAD-4(1)** (1985) 23–30.
30. S. E. Hambrusch, "Using overlap and minimizing contact points in channel routing", *Proceedings of the 21st Allerton Conference* (1983).
31. D. Zhou, "An optimal channel routing algorithm in the restricted wire overlap model", *INTEGRATION: the VLSI Journal* **9** (1990) 163–177.
32. M. Kaufmann, "Allowing overlaps makes switch-box layouts nice", *Proceedings of the* 2^{nd} *IEEE Symposium on Parallel and Distributed Processing* 1990 267–274.

Algorithmic Aspects of VLSI Layout
Eds. M. Sarrafzadeh and D. T. Lee
© World Scientific Publishing Co.

A NOTE ON THE COMPLEXITY OF STOCKMEYER'S FLOORPLAN OPTIMIZATION TECHNIQUE

TING-CHI WANG and D. F. WONG
Department of Computer Sciences, University of Texas at Austin
Austin, Texas 78712, U.S.A.

ABSTRACT

Stockmeyer in Ref. 1 presented an optimal algorithm to solve the floorplan area optimization problem in polynomial time. Although his algorithm was designed only for slicing floorplans, the technique he used can be naturally extended to solve the problem for hierarchical floorplans. In this paper, we study the worst-case time complexity of Stockmeyer's technique to solve the floorplan area optimization problem for hierarchical floorplans of order 5. We show that Stockmeyer's technique inherently has exponential worst-case time complexity for this class of floorplans. Our proof is based on the fact that we can construct examples of hierarchical floorplans of order 5 such that the number of all non-redundant implementations for any of such floorplans is exponential in the total number of modules. We also observe that if the modules all have integer dimensions, the problem can be solved in pseudo-polynomial time by Stockmeyer's technique.

Keywords: Floorplan design, floorplan area optimization, Stockmeyer's technique, slicing floorplans, hierarchical floorplans.

1. Introduction

Floorplan design[2,3,4,5,6] is an important step in the physical design of VLSI circuits. A typical approach to floorplan design is to first determine the topology (i.e., relative positions of the modules) of the floorplan primarily using the interconnection information among the modules.[3,7] After the topology of the floorplan is determined, various optimizations are then performed on it to minimize different cost measures. If each module is a rectangle with a finite set of implementations, one optimization problem is to select an appropriate implementation for each module such that the total area of the floorplan is minimized. This problem is referred to as the *floorplan area optimization problem* (or *floorplan sizing problem*).[2,4,8,9] For slicing floorplans, Stockmeyer in Ref. 1 considered this problem for the case where each module has at most two implementations (given by the module itself and its rotation), and presented an optimal polynomial time algorithm. In fact, his algorithm can also be applied to solve the case where each module has more than two implementations. Recently, we presented in Ref. 8 an optimal algorithm based on Stockmeyer's technique to solve this problem for hierarchical floorplans, and reported good experimental results.

In this paper, we study the worst-case time complexity of Stockmeyer's technique to solve the floorplan area optimization problem for hierarchical floorplans of order 5. Hierarchical floorplans of order 5 are the most natural extension of slicing floorplans and in general are non-slicing floorplans.[6,8,9] We show that Stockmeyer's technique inherently has exponential worst-case time complexity for this class of floorplans. Our proof is based on the fact that we can construct examples of hierarchical floorplans of order 5 such that the number of all non-redundant implementations for any of such floorplans is exponential in the total number of modules. We also consider the special case where the modules all have integer dimensions, and observe that this case can be solved in pseudo-polynomial time by Stockmeyer's technique.

The rest of this paper is organized as follows. In Section 2, we introduce some definitions, and formulate the floorplan area optimization problem. In Section 3, we discuss the complexity of Stockmeyer's technique for slicing floorplans, and in Section 4, we study the complexity of Stockmeyer's technique for hierarchical floorplans of order 5. Finally, we conclude this paper in Section 5 with some remarks.

2. Problem Formulation

A *floorplan* for m modules (named $1, 2, ..., m$) consists of an enveloping rectangle subdivided by horizontal and vertical line segments into m non-overlapping rectangles called *basic rectangles* (labeled $1, 2, ..., m$). Each basic rectangle i must be large enough to accommodate module i.

There are two kinds of floorplans: slicing floorplans and non-slicing floorplans. A *slicing floorplan* is a floorplan which can be obtained by recursively partitioning a rectangle into two parts by either a vertical line segment or a horizontal line segment.[1,3,5] In general, we can use a *floorplan tree* to represent the topology of a slicing floorplan. Each leaf in the tree corresponds to a basic rectangle (or a module) and each internal node corresponds to a rectangular sub-floorplan. In addition, each internal node has exactly two children and is labeled as v or h, corresponding to partitioning the associated rectangular sub-floorplan vertically or horizontally. Figure 1 shows a slicing floorplan and its corresponding floorplan tree. On the other hand, a *non-slicing floorplan* is a floorplan which is not a slicing one. A *wheel* is a non-slicing floorplan for five modules. There are only two possible wheels as shown in Fig. 2 and they are the simplest non-slicing floorplans.

A natural extension of slicing floorplans is the class of hierarchical floorplans of order 5. A floorplan is said to be *hierarchical of order 5* if it can be obtained by recursively partitioning a rectangle into two parts by either a vertical line segment or a horizontal line segment, or into five parts by a wheel.[6,8,9] (Hence, by definition, the class of slicing floorplans is a subclass of hierarchical floorplans of order 5.) Similarly, the topology of a hierarchical floorplan of order 5 can be represented by a floorplan tree, but each internal node now has either two or five children, corresponding to partitioning the associated rectangular sub-floorplan into two or

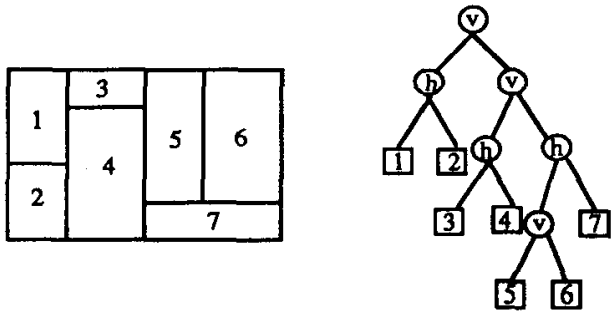


Fig. 1. A slicing floorplan and its corresponding floorplan tree.

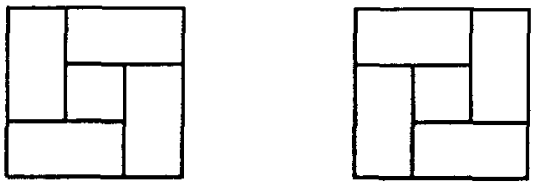


Fig. 2. Two possible wheels.

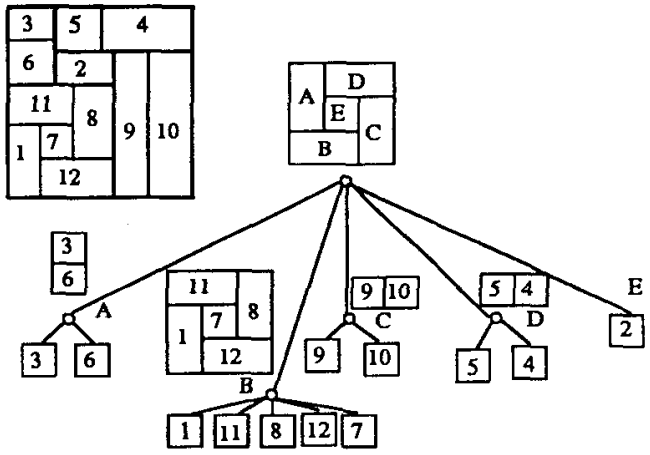


Fig. 3. A hierarchical floorplan of order 5 and its corresponding floorplan tree.

five parts. Figure 3 shows a hierarchical floorplan of order 5 and its corresponding floorplan tree.

The floorplan area optimization problem considered in this paper can be described as follows. We are given a floorplan tree specifying the topology of a hierarchical floorplan of order 5 for a set of m modules. Each module is a rectangle with a finite set of possible implementations. For each module i, we are given a set of non-

redundant implementations[a] of the form $\{(w_{i,1}, h_{i,1}), (w_{i,2}, h_{i,2}), ..., (w_{i,n_i}, h_{i,n_i})\}$, where $w_{i,j}$ is the width and $h_{i,j}$ is the height, of module i. The objective of this problem is to determine an appropriate non-redundant implementation for each module such that, without changing the topology, the total area of the floorplan is minimized.

3. Stockmeyer's Technique: Slicing Floorplans

Stockmeyer in Ref. 1 presented an optimal algorithm to solve the floorplan area optimization problem for slicing floorplans in polynomial time. Although the algorithm was designed for determining the orientations of modules (i.e., at most two non-redundant implementations for each module), it can also be applied to the more general case where each module has a finite number of non-redundant implementations.

Given a slicing tree T representing a slicing floorplan F, the algorithm begins by assigning each leaf in T the set of all non-redundant implementations of its corresponding module. The algorithm then computes the set of all non-redundant implementations for each internal node in T in a bottom-up fashion. Let v be an internal node in T with children v_1 and v_2, and let L_v, L_{v_1} and L_{v_2} be the sets of all non-redundant implementations of v, v_1 and v_2, respectively. Each L_v is computed by the algorithm directly based on L_{v_1} and L_{v_2}. Let r be the root of T. After L_r is computed, the algorithm proceeds to select an optimal implementation from L_r, and then based on this implementation, determines the optimal implementation for each of the rest of nodes in T in a top-down fashion. A key observation for the algorithm to have polynomial time complexity is stated in the following theorem. (The proof of the theorem can be found in Ref. 1, but we present here an alternative proof.)

Theorem 1. *Let v be any internal node of a slicing tree T with two children v_1 and v_2. Let L_v, L_{v_1} and L_{v_2} be the sets of all non-redundant implementations of v, v_1 and v_2, respectively. We have $|L_v| < |L_{v_1}| + |L_{v_2}|$. Thus, $|L_v|$ is less than the total number of non-redundant implementations of all the leaves in the subtree rooted at v.*

Proof. Suppose v represents a rectangular sub-floorplan partitioned by a vertical line segment into two rectangles represented by v_1 and v_2 as shown in Fig. 4(a). According to Fig. 4(a), we know the width of v is determined by the sum of the widths of v_1 and v_2, and the height of v is determined by the maximum of the heights of v_1 and v_2. Since v_1 and v_2 have $|L_{v_1}|$ and $|L_{v_2}|$ non-redundant implementations, respectively, there are at most $|L_{v_1}||L_{v_2}|$ different widths of v and at most $|L_{v_1}| + |L_{v_2}| - 1$ different heights of v, by considering all combinations of the

[a] An implementation (w, h) of a module is said to be *non-redundant* if there exists no other implementation (w', h') of that module such that $w \geq w'$ and $h \geq h'$.

non-redundant implementations of v_1 and v_2. Since $|L_{v_1}| \geq 1$ and $|L_{v_2}| \geq 1$, we have $|L_{v_1}||L_{v_2}| - (|L_{v_1}| + |L_{v_2}| - 1) = (|L_{v_1}| - 1)(|L_{v_2}| - 1) \geq 0$. Consequently, there are at most $|L_{v_1}| + |L_{v_2}| - 1$ non-redundant implementations for v since this number is determined by the minimum of the number of different widths and the number of different heights, of v (i.e., $\min(|L_{v_1}||L_{v_2}|, |L_{v_1}| + |L_{v_2}| - 1)$). Hence, $|L_v| \leq |L_{v_1}| + |L_{v_2}| - 1 < |L_{v_1}| + |L_{v_2}|$. A similar argument can be made for the case where v represents a rectangular sub-floorplan partitioned by a horizontal line segment into two rectangles represented by v_1 and v_2 as shown in Fig. 4(b). Consequently, we have shown that $|L_v| < |L_{v_1}| + |L_{v_2}|$ and the rest of the theorem immediately follows by recursively applying this inequality to v_1 and v_2. □

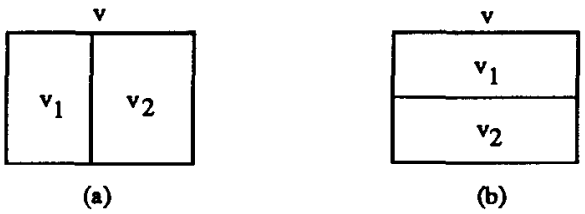


Fig. 4. Two possible cases for v.

Suppose the given slicing floorplan F has m modules and each module i has n_i non-redundant implementations. Let $k = \max_{1 \leq i \leq m}\{n_i\}$. If follows from Theorem 1 that the internal nodes in T all have less than mk non-redundant implementations, and hence computing each L_v based on L_{v_1} and L_{v_2} can be done in time polynomial in m and k. (In fact, Stockmeyer presented in Ref. 1 an efficient method to do this computation in linear time.) Since there are $m-1$ internal nodes v's in T, the total time complexity of computing all L_v's is also polynomial in m and k. Finally, it is not difficult to verify that the rest of the algorithm can be done in $O(mk)$ time, and consequently, the overall time complexity of the algorithm is polynomial in m and k.

4. Stockmeyer's Technique: Hierarchical Floorplans of Order 5

In this section, we study the time complexity of Stockmeyer's technique to solve the floorplan area optimization problem for hierarchical floorplans of order 5. Let F be a hierarchical floorplan of order 5 for m modules, and let T be the corresponding floorplan tree with $h+1$ levels. Let the root r of T be at level h and the children $v_1, v_2, ..., v_q$ of an internal node v at level i be at level $i-1$, where $1 \leq i \leq h$ and $q = 2$ or 5. Let $L_v, L_{v_1}, L_{v_2}, ..., L_{v_q}$ be the sets of all non-redundant implementations of $v, v_1, v_2, ..., v_q$, respectively. Stockmeyer's technique can be applied to T as follows.

Stockmeyer's technique
Begin
1. **for** $i = 1$ **to** h **do**
   for each internal node v at level i **do**
   Compute L_v based on $L_{v_1}, L_{v_2}, ..., L_{v_q}$.
   end for
   end for
2. Choose an implementation I with minimum area from L_r.
3. Based on I, traverse T in a top-down fashion until the optimal implementation for each leaf is determined.

End.

It is clear that Step 2 takes $O(|L_r|)$ time, and Step 3 takes $O(m)$ time since there are $O(m)$ nodes in T. However, the time complexity of Step 1 depends on the method of computing L_v.

4.1. The General Case

To see why Stockmeyer's technique has exponential worst-case time complexity for hierarchical floorplans of order 5, we need to introduce a special kind of floorplans, called nested wheels. A *nested wheel* is a hierarchical floorplan of order 5, which is obtained by initially partitioning a rectangle into a wheel and then recursively partitioning the rectangle in the middle of the wheel into another smaller wheel. Figure 5 shows a nested wheel for 13 modules and its corresponding floorplan tree.

Let F_m be a nested wheel for m modules, and let T_{F_m} be the corresponding floorplan tree. Clearly, T_{F_m} has height $h = \frac{m-1}{4}$. Note that each leaf in T_{F_m} represents a basic rectangle (or a module) and each internal node represents a wheel. Moreover, each level of T_{F_m} either consists of five leaves (at the bottom level) or consists of one internal node and four leaves (at all other levels). For each level i of T_{F_m}, $i > 0$, let v_i be the internal node and let $v_{i,1}, v_{i,2}, v_{i,3}, v_{i,4}, v_{i,5}$ be its five children with $v_{i,5}$ representing the middle rectangle of the wheel defined by v_i. Clearly, $v_{i,5} = v_{i-1}$, for all $i > 1$, and v_h is the root. (See Fig. 5 for an example.)

Given F_m, we can design the non-redundant implementations for each module such that the number of all non-redundant implementations for F_m is exponential in the total number of modules. To simplify the construction, we assume each module in F_m has two non-redundant implementations, one of which is the rotation of the other. At the beginning, two arbitrary non-redundant implementations are assigned to $v_{1,5}$. (Note that $v_{1,5}$ represents the middle module in F_m.) We then traverse the tree T_{F_m} one level at a time in a bottom-up fashion, and each time assign two non-redundant implementations to each of the four leaves at the current level and then compute the set of all non-redundant implementations for the internal node at the next higher level. Suppose we are at level $i-1$ with $i \geq 1$. Note that $v_{i,5}$ is either a leaf (if i=1) or the internal node v_{i-1} (if $i > 1$). In either case, all

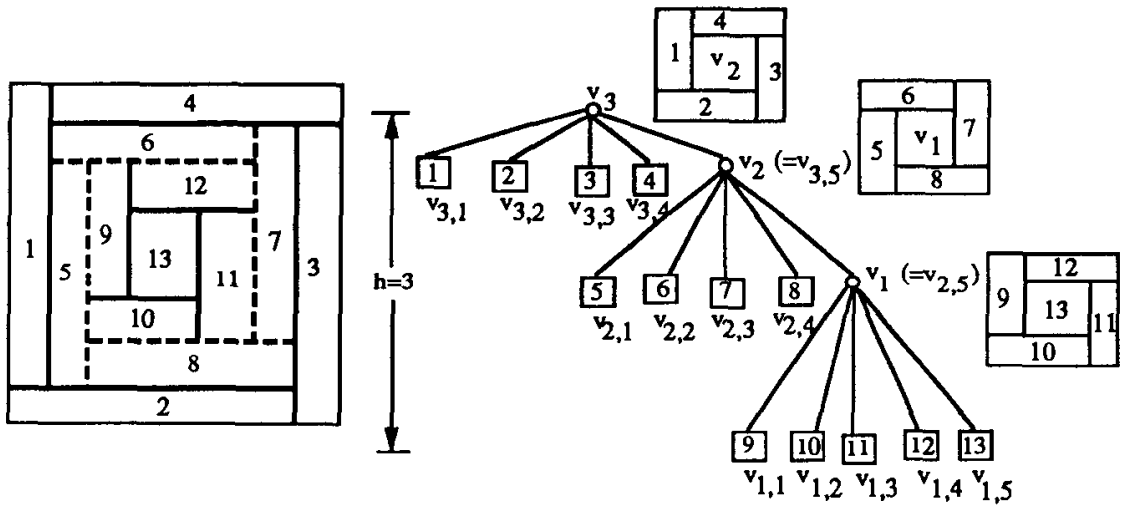


Fig. 5. A nested wheel and its corresponding floorplan tree.

non-redundant implementations of $v_{i,5}$ are known at this point. Suppose $v_{i,5}$ has k known non-redundant implementations $\{(w_1, h_1), (w_2, h_2), ..., (w_k, h_k)\}$. We may assume that

$$\forall j, 1 \leq j < k, w_j > w_{j+1} \quad \text{and} \quad h_j < h_{j+1} \ . \tag{A}$$

For each $v_{i,j}$, $1 \leq j \leq 4$, let $\{(x_j, y_j), (y_j, x_j)\}$ be its two non-redundant implementations, where x_j and y_j are positive real numbers to be determined. The following theorem gives a method to determine the values of x_j's and y_j's such that the number of all non-redundant implementations for the internal node v_i is ensured to be $2k + 1$.

Theorem 2. *If we let*

$$x_1 = x_2 = x_3 = x_4 = \max(w_1, h_k) + c_1 \ , \tag{B}$$

$$y_1 = y_2 = y_3 = y_4 = c_2 \ , \tag{C}$$

where c_1,c_2 are any arbitrary positive real numbers with

$$c_1 > c_2 \ , \tag{D}$$

then the number of all non-redundant implementations for v_i is $2k + 1$.

Proof. Suppose v_i represents the wheel as shown in Fig. 6(a). We first define the following variables.

p_1: the sum of the widths of $v_{i,1}$ and $v_{i,4}$;

p_2: the sum of the widths of $v_{i,1}$, $v_{i,5}$ and $v_{i,3}$;

p_3: the sum of the widths of $v_{i,2}$ and $v_{i,3}$;

q_1: the sum of the heights of $v_{i,1}$ and $v_{i,2}$;

q_2: the sum of the heights of $v_{i,4}$, $v_{i,5}$ and $v_{i,2}$;

q_3: the sum of the heights of $v_{i,4}$ and $_{i,3}$.

The definitions of p_1, p_2, p_3, q_4, q_5 and q_6 remain the same for the case where v_i represents the wheel as shown in Fig. 6(b). Given a non-redundant implementation chosen from each of $v_{i,1}, v_{i,2}, ..., v_{i,5}$, we can determine the values of p_1, p_2, p_3, q_1, q_2 and q_3, and then further determine the corresponding implementation of v_i. The width of v_i is determined by $\max(p_1, p_2, p_3)$, and the height of v_i is determined by $\max(q_1, q_2, q_3)$. Based on (A),(B),(C),(D) and the method of determining the implementations of v_i, for each non-redundant implementation (w_j, h_j) of $v_{i,5}$, we can determine 16 implementations of v_i by considering all combinations of the non-redundant implementations of $v_{i,1}, v_{i,2}, v_{i,3}$ and $v_{i,4}$. The 16 implementations of v_i are determined as follows.

$$(x_1, y_1), (x_2, y_2), (x_3, y_3), (x_4, y_4) \Longrightarrow (x_1 + w_j + x_3, y_4 + h_j + y_2) \quad (1)$$
$$(x_1, y_1), (x_2, y_2), (x_3, y_3), (y_4, x_4) \Longrightarrow (x_1 + w_j + x_3, x_4 + h_j + y_2) \quad (2)$$
$$(x_1, y_1), (x_2, y_2), (y_3, x_3), (x_4, y_4) \Longrightarrow (x_1 + x_4, y_4 + x_3) \quad (3)$$
$$(x_1, y_1), (x_2, y_2), (y_3, x_3), (y_4, x_4) \Longrightarrow (x_1 + w_j + y_3, x_4 + x_3) \quad (4)$$
$$(x_1, y_1), (y_2, x_2), (x_3, y_3), (x_4, y_4) \Longrightarrow (x_1 + w_j + x_3, y_4 + h_j + x_2) \quad (5)$$
$$(x_1, y_1), (y_2, x_2), (x_3, y_3), (y_4, x_4) \Longrightarrow (x_1 + w_j + x_3, x_4 + h_j + x_2) \quad (6)$$
$$(x_1, y_1), (y_2, x_2), (y_3, x_3), (x_4, y_4) \Longrightarrow (x_1 + x_4, y_4 + h_j + x_2) \quad (7)$$
$$(x_1, y_1), (y_2, x_2), (y_3, x_3), (y_4, x_4) \Longrightarrow (x_1 + w_j + y_3, x_4 + h_j + x_2) \quad (8)$$
$$(y_1, x_1), (x_2, y_2), (x_3, y_3), (x_4, y_4) \Longrightarrow (x_2 + x_3, x_1 + y_2) \quad (9)$$
$$(y_1, x_1), (x_2, y_2), (x_3, y_3), (y_4, x_4) \Longrightarrow (x_2 + x_3, x_4 + h_j + y_2) \quad (10)$$
$$(y_1, x_1), (x_2, y_2), (y_3, x_3), (x_4, y_4) \Longrightarrow (x_2 + y_3, x_1 + y_2) \quad (11)$$
$$(y_1, x_1), (x_2, y_2), (y_3, x_3), (y_4, x_4) \Longrightarrow (x_2 + y_3, x_4 + x_3) \quad (12)$$
$$(y_1, x_1), (y_2, x_2), (x_3, y_3), (x_4, y_4) \Longrightarrow (y_1 + w_j + x_3, x_1 + x_2) \quad (13)$$
$$(y_1, x_1), (y_2, x_2), (x_3, y_3), (y_4, x_4) \Longrightarrow (y_1 + w_j + x_3, x_4 + h_j + x_2) \quad (14)$$
$$(y_1, x_1), (y_2, x_2), (y_3, x_3), (x_4, y_4) \Longrightarrow (y_1 + x_4, x_1 + x_2) \quad (15)$$
$$(y_1, x_1), (y_2, x_2), (y_3, x_3), (y_4, x_4) \Longrightarrow (y_1 + w_j + y_3, x_4 + h_j + x_2) \quad (16)$$

In the above 16 formulae, the left-hand side of the arrow represents the non-redundant implementations of $v_{i,1}, v_{i,2}, v_{i,3}$ and $v_{i,4}$, respectively, and the right-hand side of the arrow represents the implementation of v_i determined by the four non-redundant implementations in the left-hand side together with the non-redundant implementation (w_j, h_j) of $v_{i,5}$. Again, based on (A),(B),(C) and (D), it is clear that any of the implementations obtained in (2)-(10) and (12)-(15) is redundant with respect to the implementation obtained in (11), and hence there exist only three non-redundant implementations out of the 16 implementations, namely,

$$(x_1 + w_j + x_3, y_4 + h_j + y_2) \quad \text{(from (1))},$$
$$(x_2 + y_3, x_1 + y_2) \quad \text{(from (11))},$$
$$(y_1 + w_j + y_3, x_4 + h_j + x_2) \quad \text{(from (16))}.$$

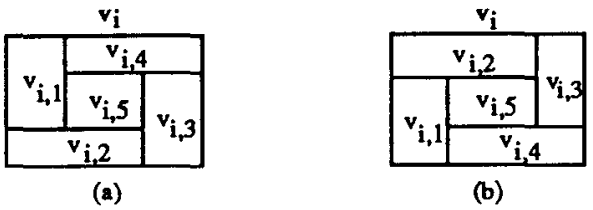


Fig. 6. Two possible cases for v_i.

If we let

$$\max(w_1, h_k) = c \ , \tag{E}$$

these three non-redundant implementations can then be rewritten as

$$(2(c + c_1) + w_j, 2c_2 + h_j) \ ,$$
$$(c + c_1 + c_2, c + c_1 + c_2) \ ,$$
$$(2c_2 + w_j, 2(c + c_1) + h_j) \ .$$

Note that the implementation obtained in (11) is independent of w_j and h_j, and hence it remains unchanged no matter what implementation of $v_{i,5}$ is chosen. Based on (A),(D) and (E), we can further prove that for any j and j' with $j' \neq j$, the implementation $(2(c + c_1) + w_{j'}, 2c_2 + h_{j'})$ (or $(2c_2 + w_{j'}, 2(c + c_1) + h_{j'})$) is not redundant with respect to $(2(c + c_1) + w_j, 2c_2 + h_j)$ and $(2c_2 + w_j, 2(c + c_1) + h_j)$, and vice versa. Consequently, after considering all combinations of the non-redundant implementations of $v_{i,1}, v_{i,2}, ..., v_{i,5}$, we can obtain all non-redundant implementations of v_i listed as follows:

$$\{(2(c + c_1) + w_j, 2c_2 + h_j) | 1 \leq j \leq k\} \cup$$
$$\{(2c_2 + w_j, 2(c + c_1) + h_j) | 1 \leq j \leq k\} \cup$$
$$\{(c + c_1 + c_2, c + c_1 + c_2)\} \ .$$

Clearly, these are exactly $2k + 1$ non-redundant implementations. □

If we arbitrarily assign two non-redundant implementations to $v_{1,5}$ (i.e., the middle module in F_m) and then apply the formulae described in Theorem 2 to design two non-redundant implementations for the rest of modules in F_m level by level in a bottom-up fashion, we have the following theorem which can be proven by induction on h.

Theorem 3. *The number of all non-redundant implementations for F_m is $3 \cdot 2^h - 1$, where h is the height of T_{Fm}. Since $h = \frac{m-1}{4}$, this number can be rewritten as*

$$(3 \cdot 2^{\frac{m-1}{4}}) - 1 = 3(2^{\frac{1}{4}})^{m-1} - 1 = \Theta(r^m)$$

for some $r > 1$. □

It follows from Theorem 3 that given a nested wheel F_m for m modules, the number of all its non-redundant implementations could be exponential in m. Since Stockmeyer's technique needs to compute the set of all non-redundant implementations for the entire floorplan before determining an optimal solution, we can deduce the following theorem.

Theorem 4. *Stockmeyer's technique inherently has exponential worst-case time complexity to solve the floorplan area optimization problem for hierarchical floorplans of order 5.* □

Remark 1. Note that F_m (constructed by our method) has a very skewed floorplan tree, i.e., $h = \Theta(m)$. In fact, we can also construct floorplans with short floorplan trees to show that Stockmeyer's technique still inherently has exponential worst-case time complexity. Given any ϵ, $0 < \epsilon < 1$, we can extend our method to construct a floorplan $F_m(\epsilon)$ with the height of its floorplan tree equal to $\Theta(m^\epsilon)$ as follows. We first use our method to construct a nested wheel $F_{m'}$ such that it has m' modules ($m' \leq m$) and the height of its floorplan tree is $\Theta(m^\epsilon)$. If $m' < m$, we then replace an arbitrary module i in $F_{m'}$ by a slicing floorplan $F(i)$ (consisting of $m - m' + 1$ modules) to get a new floorplan $F_m(\epsilon)$ for m modules. Let $T_{F(i)}$ and $T_{F_m(\epsilon)}$ be the floorplan trees representing $F(i)$ and $F_m(\epsilon)$, respectively. In our construction, we always keep $T_{F(i)}$ as balanced as possible, and hence its height is $\Theta(\log(m - m' + 1))$. Therefore, $T_{F_m}(\epsilon)$ has height bounded by $\Theta(m^\epsilon) + \Theta(\log(m - m' + 1)) = \Theta(m^\epsilon)$. Since it is not difficult to verify that the two non-redundant implementations (one of which, this time, may not necessarily be the rotation of the other) for each module in $F(i)$ can be always designed in such a way that $F(i)$ preserves the same non-redundant implementations as the replaced module i, we get the number of all non-redundant implementations for $T_{F_m}(\epsilon)$ equal to $\Theta(r^{m^\epsilon})$ for some $r > 1$.

4.2. A Special Case

We now consider a special case where the modules all have integer dimensions. Let L be the maximum dimension among the modules. Lengauer in Ref. 2 (page 347) pointed out that if the given floorplan for m modules is specified by an unoriented slicing floorplan tree, the number of all non-redundant implementations of each internal node in the tree is no more than mL. His argument can also be applied to hierarchical floorplans of order 5 to prove the following theorem.

Theorem 5. *Given any floorplan tree specifying the topology of a hierarchical floorplan of order 5 for m modules, if the modules all have integer dimensions no greater than L, then each internal node in the tree has no more than mL non-redundant implementations.* □

Based on Theorem 5, it is easy to see that we can compute all the non-redundant implementations for each internal node in the given floorplan tree in time polynomial in m and L. Thus, we have the following theorem.

Theorem 6. *The floorplan area optimization problem for hierarchical floorplans of order 5 can be solved in pseudo-polynomial time by Stockmeyer's technique if the given modules all have integer dimensions.* □

Remark 2. Theorem 5 still remains true for any hierarchical floorplans of order k, where $k > 5$. (A floorplan is said to be *hierarchical of order k* if it can be obtained by recursively partitioning a rectangle into at most k parts.) In addition, Theorem 6 also remains true for any hierarchical floorplan of order k as long as k is a constant.

5. Concluding Remarks

In this paper, we have shown that Stockmeyer's technique inherently has exponential worst-case time complexity to solve the floorplan area optimization problem for hierarchical floorplans of order 5. Our proof is based on the fact that we can construct examples of hierarchical floorplans of order 5 such that the number of all non-redundant implementations for any of such floorplans is exponential in the total number of modules. Fortunately, this kind of examples does not represent typical floorplans that are encountered in practice. In fact, we presented in Ref. 8 an optimal algorithm based on Stockmeyer's technique to solve the problem for hierarchical floorplans and the experimental results show that our algorithm performs very well. We also observed in this paper that the special case where the modules all have integer dimensions can be solved in pseudo-polynomial time by Stockmeyer's technique.

Although Stockmeyer has shown in Ref. 1 that the floorplan area optimization problem for general floorplans is NP-hard, the floorplans used in his proof are not hierarchical floorplans of order 5. Hence, whether the NP-hard result remains true for hierarchical floorplans of order 5 is still an open problem.

Acknowledgements

This work was partially supported by the National Science Foundation under grant MIP-8909586, by an IBM Faculty Development Award, and by an ACM SIGDA scholarship.

References

1. L. Stockmeyer, "Optimal orientations of cells in slicing floorplan designs", *Information and Control* **Vol. 59** (1983) 91–101.
2. T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout* (Wiley-Teubner, 1990).
3. R. H. J. M. Otten, "Automatic floorplan design", *Proc. 19th ACM/IEEE Design Automation Conf.*, 1982, pp. 261–267.

4. M. Pedram and B. Preas, "A hierarchical floorplanning approach", *Proc. IEEE International Conf. on Computer Design*, 1990, pp. 332–338.
5. D. F. Wong and C. L. Liu, "A new algorithm for floorplan design", *Proc. 23rd ACM/IEEE Design Automation Conf.*, 1986, pp. 101–107.
6. D. F. Wong and K.-S. The, "An algorithm for hierarchical floorplan design", *Proc. IEEE International Conf. on Computer-Aided Design*, 1989, pp. 484–487.
7. U. Lauther, "A min-cut placement algorithm for general cell assemblies based on a graph representation", *Journal of Digital Systems* **Vol. IV** (1980) 21–34.
8. T.-C. Wang and D. F. Wong, "An optimal algorithm for floorplan area optimization", *Proc. 27th ACM/IEEE Design Automation Conf.*, 1990, pp. 180–186.
9. D. F. Wong and P. Sakhamuri, "Efficient floorplan area optimization", *Proc. 26th ACM/IEEE Design Automation Conf.*, 1989, pp. 586–589.

Algorithmic Aspects of VLSI Layout
Eds. M. Sarrafzadeh and D. T. Lee
© World Scientific Publishing Co.

AN ALGORITHM TO ELIMINATE ALL COMPLEX TRIANGLES IN A MAXIMAL PLANAR GRAPH FOR USE IN VLSI FLOORPLAN

SHUJI TSUKIYAMA
Department of Information and System Engineering, Chuo University, 1-13-27 Kasuga Bunkyo-ku, Tokyo 112, Japan

KEIICHI KOIKE
LSI Laboratories, NTT Corporation, 3-1 Morinosato Wakamiya Atsugi-shi, Kanagawa 243-01, Japan

and

ISAO SHIRAKAWA
Department of Information Systems Engineering, Osaka University, 2-1 Yamadaoka Suita-shi, Osaka 565, Japan

ABSTRACT

A rectangular dual D of a planar graph G is a planar embedding of a dual graph of G such that all inner faces and the total graph enclosure are rectangular. Regarding each inner face of a rectangular dual as an area for a functional block to be placed, we can use the rectangular dual for a floorplan of a VLSI. If a planar graph G representing a given VLSI circuit contains a complex triangle (a cycle of length three which is not a boundary of a face), then G does not have a rectangular dual. Therefore, such complex triangles must be eliminated from G.

This paper proposes a heuristic algorithm to this problem, which is to find a minimal set E^* of edges such that all complex triangles of a given planar graph are eliminated by changing each edge of E^* to two serial edges. The algorithm has the time complexity of $O(|V|^2)$ and a constant worst-case performance ratio of 2, where V is a set of vertices.

Keywords: Complex triangle, maximal planar graph, rectangular dual, heuristic algorithm, VLSI floorplan.

1. Introduction

With recent progress in the technology of VLSI fabrication, demands for general cell (or macro cell) VLSI's are increasing rapidly. Usually, such general cell VLSI's are designed by using hierarchical top-down strategies. Therefore, the floorplanning to determine the shape and the relative position of each functional block (or macro) is important for minimizing chip area, since each block can be realized so as to achieve the global optimality.

Given a planar graph G = [V ∪ {X}, E], a planar embedding D of a dual graph of G such that each face and the total graph enclosure are rectangles and the exterior face of D corresponds to a specific vertex X is called a **rectangular dual**[1,2,3,4,5] of G (see Figs. 1 and 2). Regarding each inner face of a rectangular dual as an area for a block to be placed, we can use the rectangular dual for a floorplan of general cell VLSI. Such a floorplanning technique using the rectangular duals seems to be one of the fundamental approaches to the VLSI hierarchical design, since in order to determine the shape and the relative position of each functional block, the technique uses more global information concerning not only interconnection requirements among functional blocks but also the total chip area, in comparison with the so-called force-directed[6] or min-cut[7] approach.

In the technique, a given circuit is to be represented by a planar graph G = [V ∪ {X}, E] with weights on all vertices of V and all edges. Vertex X represents the set of I/O pads, and each vertex of V and its weight correspond to a functional

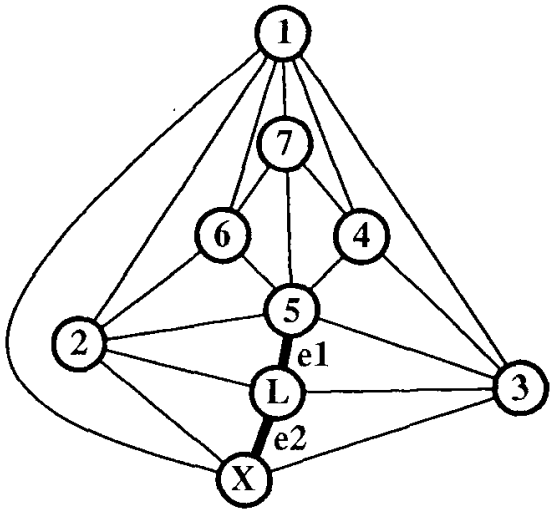


Fig. 1. A maximal planar graph G, which is obtained from Fig. 5 by introducing link vertex L and new edges e1 and e2.

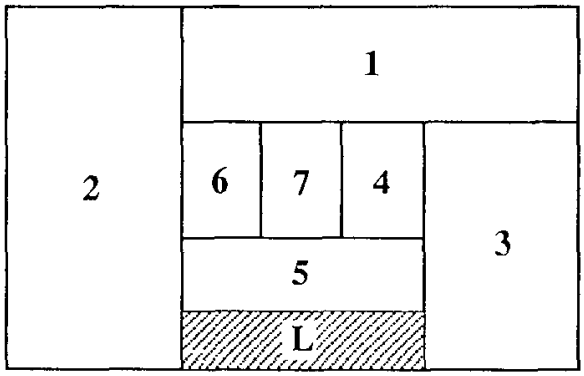


Fig. 2. A rectangular dual of graph G shown in Fig. 1.

block and the area required to realize the block, respectively. Each edge and its weight represent interconnection requirements and the number of wires between two functional blocks corresponding to the end vertices of the edge, respectively.

As can be seen from the definition, the degree of a vertex in a rectangular dual D must be three or four. However, if we regard a vertex of degree four as two vertices of degree three which are located at the same point, then we can see that the degree of each vertex in D is three.[5] Therefore, in order that a planar graph G has a rectangular dual D, each face of G must be bounded by three edges, that is, G must be **triangulated**. A cycle consisting of three edges is a **triangle**, and a planar (triangulated) graph such that each face is bounded by a triangle is called a **maximal planar graph**, since no edge can be added to it without violating planarity.

A triangle of a maximal planar graph which is not a boundary of a face is called a **complex triangle**[2] or a **separating triangle**.[8] If a maximal planar graph $G = [V \cup \{X\}, E]$ representing a given circuit has a complex triangle, then G does not have a rectangular dual.[2] Therefore, in order to construct a rectangular dual from G, it is necessary to eliminate complex triangles from G. In this paper, we consider a problem of eliminating all complex triangles from a maximal planar graph, and propose a heuristic algorithm of time complexity $O(|V|^2)$ with a constant worst-case performance ratio not greater than 2. The preliminary version of this paper was presented in Ref. 9, and recently, this problem is shown to be strongly NP-complete.[10]

The basic graph theoretic terminologies are used without definitions in the following (for example, see Ref. 11 for definitions).

2. Floorplanning System

Let us show briefly basic ideas of our floorplanning system using the rectangular dual,[3,12] in order to understand the formulation of our problem.

Input to our floorplanning system is a **block diagram** B as shown in Fig. 3. For each **(functional) block** b shown by a rectangle, weights A(b), R(b), and r(b) are specified, where A(b) is the area of a rectangle realizing block b, and R(b) and r(b) are the upper and lower bounds to the aspect ratio of a rectangle with area A(b) realizing block b. For each inter-block **net** n, the **bundle size** w(n) is specified as a weight which indicates the number of wires in net n. Moreover, the upper bound R_{chip} to the aspect ratio of the total chip area is given as input.

Usually, a floorplanning system using the rectangular dual consists of the following two phases.

I. For a given block diagram B, construct a maximal planar graph $G = [V \cup \{X\}, E]$ such that

   1: the set of I/O pads of B corresponds to a specific vertex X,
   2: each block v of B corresponds to a vertex v of V,

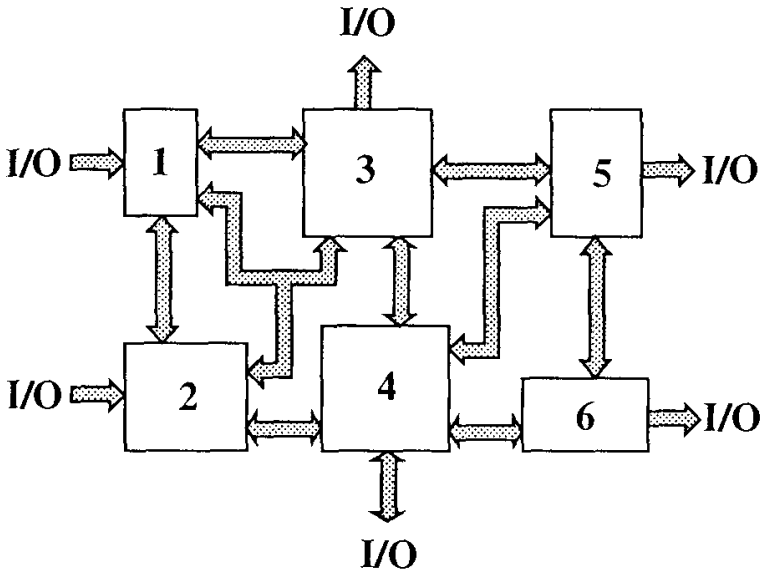


Fig. 3. Block diagram B.

3: each net n of B corresponds to a set of those edges which span vertices corresponding to the blocks connecting to n, (therefore, the bundle size w(e) of an edge e of G may be the sum of the bundle sizes of the nets corresponding to edge e), and
4: G has a rectangular dual, which can be drawn in as small an area as possible under constraints C1, C2 and C3 stated below.

II. From among all rectangular duals D of the maximal planar graph G obtained above, find a most area-efficient drawing of D such that

C1: each face of the drawing corresponding to a block v is large enough to admit the rectangle of area A(v) and of aspect ratio between r(v) and R(v),
C2: the length of each edge e separating two abutting faces is not less than the bundle size w(e) of the edge e of G, and
C3: the aspect ratio of the total chip area is not greater than R_{chip}.

The second phase contains a subproblem of determining the shape of each block for a given relative position so as to minimize the total chip area. On this problem, much research has been done. In the case where the aspect ratio of the face for each block is not restricted, the problem can be solved efficiently[13]; otherwise, it is an NP-hard problem.[14] However, even if the aspect ratio is restricted, the problem can be solved in polynomial time when the relative position of each block has the so-called slicing structure.[15] Since we must impose the constraints C1 and C2 stated above from a practical viewpoint, the second phase contains a NP-hard subproblem. Therefore, we attacked this phase by using a sort of Branch-and-Bound technique.[16]

For the first phase, few research has been done, although a necessary and sufficient condition for a planar graph to have a rectangular dual has been found.[2] Ref. 17 considers this problem, but does not construct a planar graph G such that G has a rectangular dual whose drawing has as small an area as possible. Once a graph G representing B is constructed, a set of the rectangular duals whose drawings are considered in the second phase is restricted. Namely, a certain set of relative positions of blocks is determined by constructing G, and from among the set of relative positions, the one yielding the most area-efficient floorplan will be selected in the second phase. Therefore, it is important to construct a planar graph G with a rectangular dual which can be drawn in as small an area as possible.

Concerning the problem of determining the relative position of each block, several algorithms have been used such as a forth-directed placement method,[6] a min-cut placement method,[7] an enumerative method,[18] and so on. The main difficulty of this problem is the lack of universal criteria in the determination of the relative position of each block, since it is hard to estimate the final chip area precisely in this stage. Therefore, we adopted a heuristic manner in this phase.

In a floorplanning system using the rectangular dual, if graph G has an edge e = (u,v), then faces u and v corresponding to vertices u and v of G will be abutted each other in a drawing of D. Hence, in order to generate an area-efficient floorplan, each vertex v of G must be connected with other vertices by edges, in such a way that the total lengths of the edges in a drawing of D separating face v from the faces abutting v is close to the length of perimeter of a rectangle with area A(v). Namely, many vertices can be connected by edges with a vertex corresponding to a large block but not with a vertex corresponding to a small block. The first phase consists of the following steps, in which we try to achieve this property in a heuristic manner.

I-1 [Multi-Terminal Net Reduction]: Since a given block diagram may have a multi-terminal net, it cannot be represented by a graph straightforwardly. Therefore, we replace each multi-terminal net with a set of edges spanning those vertices which correspond to the blocks connecting to the multi-terminal net.

I-2 [Planarization]: If graph G obtained by I-1 is non-planar,[19] then we delete some edges from G by using a min-cut method[20] recursively, so that G becomes planar. For each deleted edge e, we find a path consisting of edges of planar graph G, and add the bundle size w(e) to the bundle size of each edge on the path. (This path indicates a route in a VLSI chip for connecting the net corresponding to edge e.)

I-3 [Embedding]: If this G is triconnected,[21] then the planar embedding of G such that the specific vertex X is on the outer face is determined uniquely, exclusive of the embedding of edges incident with X. Otherwise, planar embedding of G is not unique, and hence we determine a planar embedding of G.

I-4 [Triangulation]: For a face bounded by more than 3 edges in the planar embedding, we select a pair of vertices v and u, and add edge (v,u) with the bundle size equal to 0, until G becomes a maximal planar graph.

I-5 [Elimination of Complex Triangle]: If maximal planar graph G has no rectangular dual, then we modify G to a graph with at least one rectangular dual.

3. Complex Triangle Elimination

Let us formulate the problem to be solved in Step I-5. A necessary and sufficient condition for a maximal planar graph G to have a rectangular dual D can be stated as follows.

An edge between two vertices on a cycle which is not contained in the cycle is called a **chord** of the cycle. Let S be the cycle consisting of vertices adjacent to a specific vertex X in a maximal planar graph G. Given a chord e = (v, w) of S, there exist exactly two paths on S between v and w. If one of these two paths has no end vertices of other chord of S, then the chord e = (v, w) is called a **critical chord** of G, and such a path on S between v and w is called a **corner implying path.**[2]

Theorem 1.[2] *A maximal planar graph $G = [V \cup \{X\}, E]$ has a rectangular dual D, if and only if G satisfies the following two conditions;*

1: there exists no complex triangle not containing vertex X, and

2: there do not exist more than four critical chords.

If there exists a critical chord e = (v, w) of S, then from among vertices on the corner implying path between v and w (exclusive of v and w), we have to choose at least one **corner vertex** cv such that the corresponding face of D is located on a corner (see Fig. 4). Hence, we cannot choose four corner vertices from vertices on S unrestrictedly, so that we may not be able to optimize the area and the aspect ratio of the chip. Thus, we modify a maximal planar graph G to the one containing no critical chord. Noting that a critical chord e = (v, w) forms a complex triangle {v, w, X} together with edges (v, X) and (w, X), the problem of eliminating all critical chords is reduced to the one of eliminating all complex triangles containing vertex X. Thus, in our system, we eliminate all the complex triangles in G.

If a maximal planar graph G contains a complex triangle, it is eliminated as follows: Choose an edge of the complex triangle, and split the edge into two edges by introducing a new vertex called a **link vertex** L on the edge. Then, add two new edges incident with L in order to keep the graph being a maximal planar graph (see Figs. 5 and 1). We can easily see that this process does not create a new complex triangle. Therefore, repeated application of this process eliminates all complex triangles.

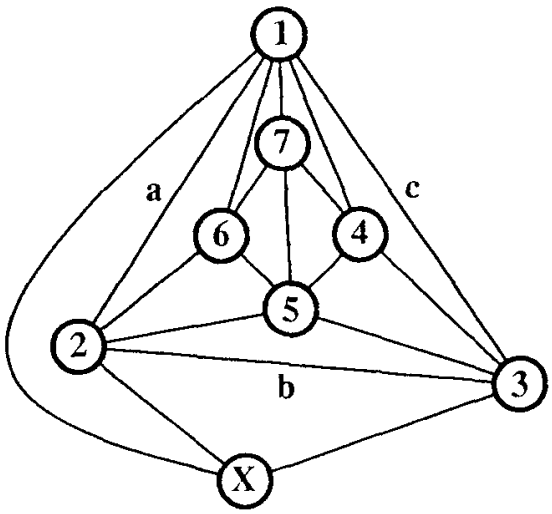


Fig. 4. A maximal planar graph G with complex triangle {a,b,c}.

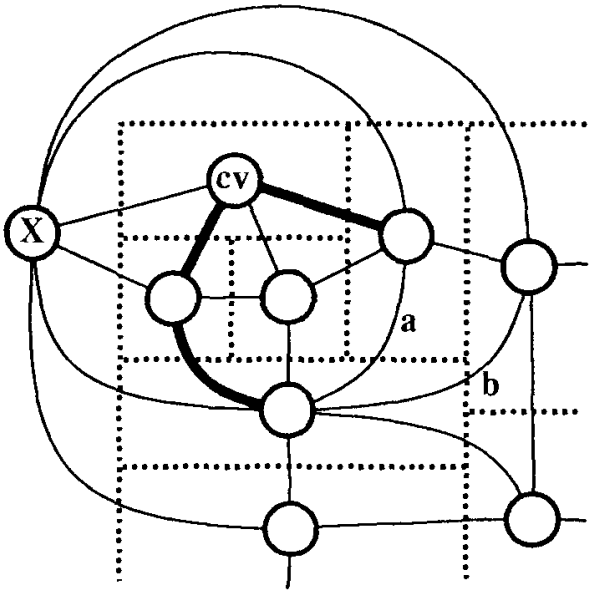


Fig. 5. Critical chord a, chord b, corner vertex cv, and corner implying path (bold lines).

As seen from Fig. 2, if we introduce a link vertex L on edge b = (2,3) in G, a rectangular dual of G has a face corresponding to L. This face is not used actually for a functional block in VLSI chip, but it is supposed to be of possible use for the interconnection between two functional blocks corresponding to the end vertices 2 and 3 of the edge b. Therefore, the minimization of such an area contributes to the reduction of the whole chip area. Thus, the problem of our interest can be formulated as follows.

[Complex Triangle Elimination Problem]: *Given a maximal planar graph $G = [V \cup \{X\}, E]$ with weights $w(\cdot)$ on its edges, find a set E^* of edges such that E^* contains at least one edge of every complex triangle in G and the sum $w(E^*)$ of the weights of edges in E^* is minimum.*

To attack this problem, we must detect all the complex triangles of G. This can be done easily in time of $O(|V|^2)$, if a planar embedding of G is given by appropriate adjacency lists. For example, a planar embedding of G can be represented by an adjacency list of each vertex v, which contains vertices w adjacent to v in the order of appearance of edge (v, w) when we explore the planar embedding clockwise around v.[19] Henceforth, we assume that a planar embedding of a graph is specified in such a manner.

Let T be a set of complex triangles of G = [V ∪ {X}, E], and let G(T) = [V(T), E(T)] be a subgraph of G consisting of vertices and edges contained in complex triangles of T. Then, there hold

$$|T| \leq |V| - 3 \quad \text{and} \tag{1}$$

$$|E(T)| \leq 2|V| - 5 \ . \tag{2}$$

It is not difficult to verify these inequalities by induction of |V|. Moreover, there exists a case where both equalities hold.

If a subset E′ of E(T) contains at least one edge of every triangle t of T, then E′ is said to be a **cover** of T. If a cover E* of T satisfies $w(E^*) \leq w(E')$ for any cover E′, then E* is called a **minimum cover** of T, where w(E′) is the sum of weights w(e) of edges e in E′. Then, we can rewrite our problem as follows.

[Problem CTE]: *Given a planar embedding of $G(T) = [V(T), E(T)]$ with weights $w(\cdot)$ on its edges and a set T of specified triangles of $G(T)$, find a minimum cover E^* of T.*

When we considered this problem,[9] we did not know whether or not the problem is NP-hard, but recently, the decision problem version of this problem is proven to be strongly NP-complete by showing a transformation from Planar 3-Dimensional Matching Problem.[10]

4. Covering Algorithms

Since Problem CTE can be formulated as Set-Covering Problem[22] straightforwardly, we can solve a given instance of Problem CTE as an instance of Set-Covering Problem. However, before doing this, we can reduce the size of the problem, as shown in the following.

If there exists a triangle t of T such that each edge of t does not belong to any other triangle of T, *i.e.*, t is a biconnected component of G(T) by itself, then a minimum cover E* of Problem CTE contains an edge e of t with the minimum

weight among edges of t. Moreover, if among edges e_1, e_2, and e_3 of a triangle t of T, only one, say e_1, belongs to other triangle in T, and the other two, e_2 and e_3, belong to only t, then we can prove the following propositions.

Proposition 1. *Let t and e_1 be such triangle and edge stated above, respectively, and if the weight $w(e_1)$ is minimum among edges of t, then there exist a minimum cover E^* containing e_1.*

Proof. Suppose that E^* does not contain e_1, and E^* contains either e_2 or e_3. If E^* contains e_2, then cover $E' = (E^* - \{e_2\}) \cup \{e_1\}$ satisfies $w(E') \leq w(E^*)$, and hence E' is also a minimum cover. □

Thus, in the case when the condition in the proposition is satisfied, we can construct a minimum cover E^* by adding e_1 to a minimum cover E_1 of Problem CTE on $G(T_1) = [V(T_1), E(T_1)]$, where T_1 is obtained from T by deleting all triangles containing e_1. Therefore, our problem on G(T) is reduced to the one on $G(T_1)$.

In the case when the weight $w(e_1)$ is not minimum among edges of t, we can reduce our problem to the one on a graph $G(T'') = [V(T''), E(T'')]$, weight function $w''(\cdot)$, and T'', such that $T'' = T - \{t\}$, $E(T'') = E(T) - \{e_2, e_3\}$, and

$$w''(e) = \begin{cases} w(e) - \min[w(e_2), w(e_3)]; & \text{if } e = e_1, \\ w(e); & \text{otherwise .} \end{cases} \tag{3}$$

Proposition 2. *Let E'' be a minimum cover of Problem CTE on $G(T'')$, $w''(\cdot)$, and T'', and assume e_2 be the edge with the minimum weight $w(e_2)$ among edges of t, then the following set E^**

$$E^* = \begin{cases} E''; & \text{if } e_1 \text{ belongs to } E'', \\ E'' \cup \{e_2\}; & \text{otherwise ,} \end{cases} \tag{4}$$

is a minimum cover of Problem CTE on $G(T), w(\cdot)$, and T.

Proof. Since there holds $w(E^*) = w''(E'') + w(e_2)$, we can easily verify the proposition from the fact that e_2 has the minimum weight among edges of t and E^* must contain at least one edge of t. □

By applying Propositions 1 and 2 repeatedly, we can reduce the size of the problem (the size of graph G(T) and the number of triangles in T). The total time spent by this process is $O(|V|)$. At the termination of this process, each triangle t of T contains at least two edges each of which belongs to a triangle of T other than t. After this reduction, we consider Problem CTE as a Set-Covering Problem.

If the size of G(T) is reduced sufficiently by these propositions, we can adopt a Branch-and-Bound method to solve Set-Covering Problem by using a penalty

function ($cost^{22}$) as a lower bound to the weight w(E′) of a cover E′. In many practical cases, we can solve Problem CTE in this manner.

If the size of G(T) is not small, we can solve Problem CTE with the use of the heuristic algorithm in Ref. 22, which has the time complexity of $O(|V|^2)$ and the worst-case performance ratio of $\log |V|$, where the worst-case performance ratio is the worst-case ratio of $w(E_{alg})$ of a cover E_{alg} found by the algorithm to $w(E^*)$ of a minimum cover. However, if we formulate Problem CTE as Set-Covering Problem, we may lose a certain property that triangles are included in a planar graph. Hence, we cannot use the planarity of triangles, which may help us to devise a good algorithm. Thus, in the case when the size of G(T) cannot be reduced sufficiently by Propositions 1 and 2, we solve the problem in a different manner.

5. Heuristic Algorithm

If G(T) is not biconnected, we can consider Problem CTE on each biconnected component of G(T) separately. Therefore, without loss of generality, we may assume that G(T) is biconnected.

For distinct triangles t_1 and t_2 of T, we say that t_1 **contains** t_2, denoted by $t_2 \subset t_1$, if $F(t_2)$ is contained in $F(t_1)$, where F(t) for a triangle t is a region bounded by edges of t in a given planar embedding of G(T). For an edge e in G(T), let $t_{max}(e)$ be a triangle of T with edge e such that there exists no triangle of T with e containing $t_{max}(e)$, and let $t_{min}(e)$ be a triangle of T with edge e which contains no other triangle of T with e. From these definitions and that of G(T), we see that for any edge e there exists at least one triangle $t_{max}(e)$ ($t_{min}(e)$), and there exist at most two distinct triangles $t_{max}(e)$ ($t_{min}(e)$). Moreover, if $t_{min}(e)$ is uniquely defined for an edge e, then $t_{max}(e)$ is also unique for this edge e, and $t_{max}(e)$ contains $t_{min}(e)$ unless $t_{max}(e) = t_{min}(e)$.

With the use of these notations, we introduce a **rooted tree** R = [Tr, Br] with **root** r as follows:

1: Each vertex t of Tr other than r corresponds to a triangle t of T, *i.e.*, Tr = T ∪ {r}.
2: There exists an edge (t_1, t_2) of Br from t_1 to t_2 ($t_1, t_2 \in T$ and $t_1 \neq t_2$), if and only if t_1 immediately contains t_2, that is, $t_2 \subset t_1$ and there exists no triangle t of T such that $t_2 \subset t \subset t_1$.
3: There exists an edge (r,t) of Br from r to t, if and only if there exists no triangle of T containing t.

Since we can easily see that R is a rooted tree with root r, a **son**, a **father**, and a **descendant** of a vertex in R are defined in a usual manner.

For this R, we introduce a set C of edges between two distinct vertices of Tr as follows:

1: Each edge e of C corresponds to an edge e of E(T), and has a weight w(e) equal to the weight of the corresponding edge e of E(T).

2: If for edge e of E(T) there exist two distinct $t_{min}(e)$, say t_1 and t_2, then the corresponding edge e of C is incident with the vertices of Tr corresponding to t_1 and t_2.

3: If for edge e of E(T) there exists a unique $t_{min}(e)$, then the corresponding e of C is incident with the vertex corresponding to $t_{min}(e)$ and the father of the vertex corresponding to $t_{max}(e)$.

Figure 6 (b) shows the rooted tree R and edges in C associated with G(T) and T shown in Fig. 6 (a). We can easily construct such R and C in time of $O(|V|^2)$,

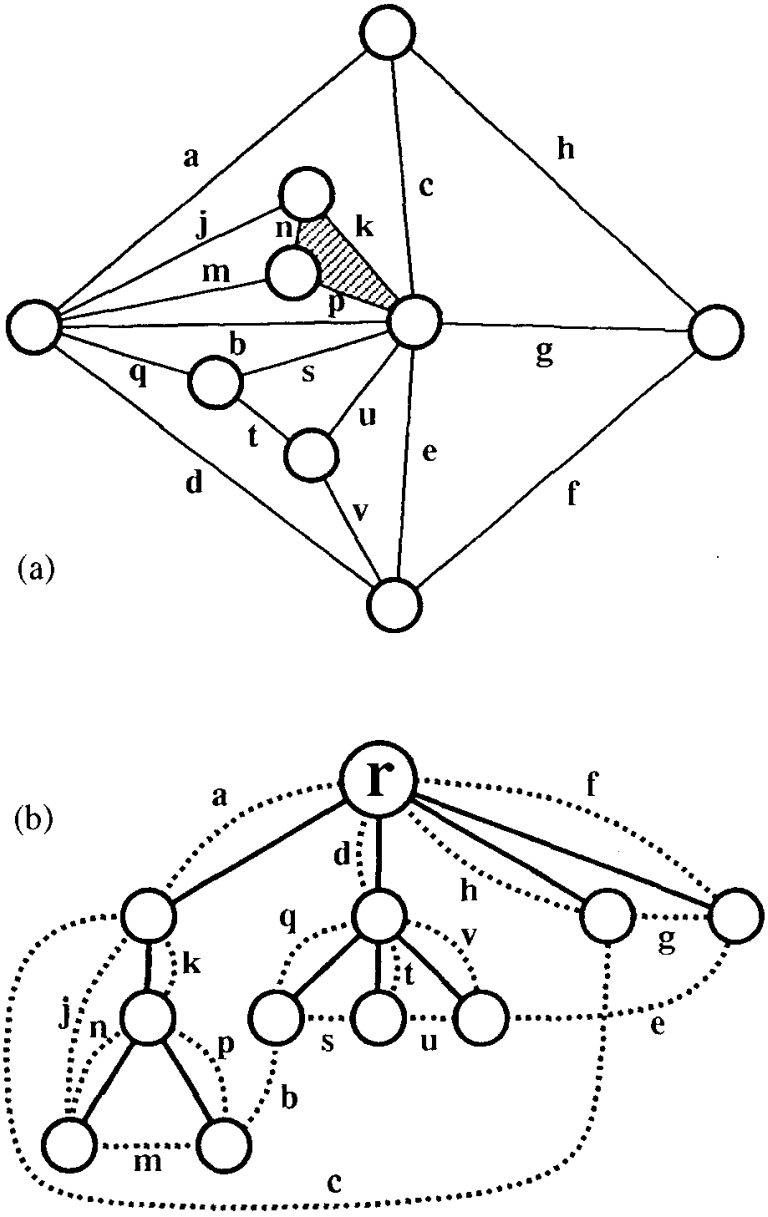


Fig. 6. (a) Graph G(T), where T contains all triangles in G(T) exclusive of the shaded one. (b) Rooted tree R (bold lines) and edges of C (dashed lines) associated with (a).

if a planar embedding of G(T) is given with the use of the adjacency lists stated before.

For any vertex t of Tr, let C(t) be a subset of C such that each edge in C(t) connects a descendant of t and a vertex which is not a descendant of t, where t itself is a descendant of t. Then, noting that triangles t of T are contained in a planar graph G(T) and at least two edges of each triangle t belong to triangles other than t, we can easily see that the following claims hold:

1: There is no parallel edges in C.
2: For any vertex t in R, C(t) consists of three edges corresponding to the edges contained in the triangle corresponding to t.
3: For any pair (t_1, t_2) of distinct sons of a vertex t in R, the number of edges contained commonly in both $C(t_1)$ and $C(t_2)$ is at most one.

Moreover, we can prove the following proposition.

Proposition 3. *Let C' be a subset of C such that a graph $R+C' = [Tr, Br \cup C']$ obtained by adding edges in C' to $R = [Tr, Br]$ is* **2-edge-connected** *(or* **bridge-connected***), then the corresponding set $E' = C'$ is a cover of T and vice versa, where 2-edge-connected means that at least two edges should be opened so as to disconnect the graph.*

Proof. Suppose that $E' = C'$ is not a cover of T, and there exists a triangle t of T such that any edge of t does not belong to E'. Then, we see from claim (b) that C' also does not contain any edge of C(t). This means that edge (f(t),t) of Br connecting vertex t and its father f(t) is a **bridge**, that is, if edge (f(t),t) is deleted from R + C', then R + C' is disconnected. This is a contradiction.

On the other hand, suppose that for a cover E' of T, the addition of the corresponding set $C' = E'$ to rooted tree R does not change R to be 2-edge-connected. Then, there exists a bridge $(f(t), t) \in Br$ in R + C', that is, there exists a vertex t such that C' does not contain any edge of C(t). From claim 2, this means that E' does not contain any edge of t, which is a contradiction. □

As can be seen from this proposition, Problem CTE is transformed to a problem of finding a subset C* of C such that R + C* is 2-edge-connected and the sum w(C*) of the edge weights in C* is minimum. This problem of finding such C* is called an **Augmentation Problem**, and known to be NP-complete unless weights of all edges are the same.[23] However, there exists an approximation algorithm such that the time complexity is $O(|V|^2)$ and the worst-case performance ratio of the weight $w(C_{alg})$ of C_{alg} found by the algorithm to the weight w(C*) of an optimum solution is at most two.[23] Therefore, we can solve our problem with the use of this algorithm. The overall algorithm is described as follows. It is not difficult to see that the total time complexity is $O(|V|^2)$.

input: planar embedding of a biconnected graph G(T) = [V(T), E(T)], weights w(·) of edges, and a set T of triangles in G(T). If G(T) is not biconnected, then apply this algorithm to each biconnected component.

output: a minimal cover E* of T.

1: Apply Propositions 1 and 2 repeatedly as many as possible, in order to reduce the size of the problem. Assume that the reduced problem is specified by G(T′) = [V(T′), E(T′)], w′(·), and T′.
2: If the size of G(T′) is small enough to be solved by an exhaustive method, then consider the problem as a Set-Covering Problem and solve it by Branch-and-Bound method.[22] Let E′ be a cover of T′ found by this method, and go to step 5.
3: Otherwise, construct rooted tree R = [Tr, Br] and set C of edges from G(T′) and T′ obtained in step 1. Each edge of C corresponds to an edge of E(T′), and the weights of edges in C is the same as w′(·).
4: By the approximation algorithm in Ref. 24, find a subset C′ of C such that R + C′ is 2-edge-connected. Let E′ be the set of edges in E(T′) corresponding to those of C′.
5: Based on the ways indicated in Propositions 1 and 2, construct a cover E* of T from E′.

6. Conclusion

In this paper, we have considered a floorplanning system using the rectangular dual, and formulated a problem of eliminating all complex triangles in a maximal planar graph G = [V ∪ {X}, E] as the problem of finding a set E* of edges in G such that E* contains at least one edge from every complex triangle and the sum of the weights of edges in E* is minimum, so as to use the problem in the floorplanning system. Then, we have proposed a heuristic algorithm to this problem such that the time complexity is proportional to the square of the number of vertices, and the worst-case performance ratio is not greater than two.

Our floorplanning system has been programmed in Fortran77 and applied to a few real circuits. Figure 7 shows a result for a circuit with 13 blocks and 52 nets, in which 7 link vertices are introduced to eliminate complex triangles. The total CPU-time needed to obtain this result (including Phases I and II) is 17.8 seconds by NEC ACOS-2000 main frame computer.

Acknowledgments

The authors would like to express their appreciation to Mr. K. Tani, NEC Corporation, for his effort to complete our floorplanning system, and to Prof. S. Ueno, Tokyo Institute of Technology, for his appropriate suggestion on the references.

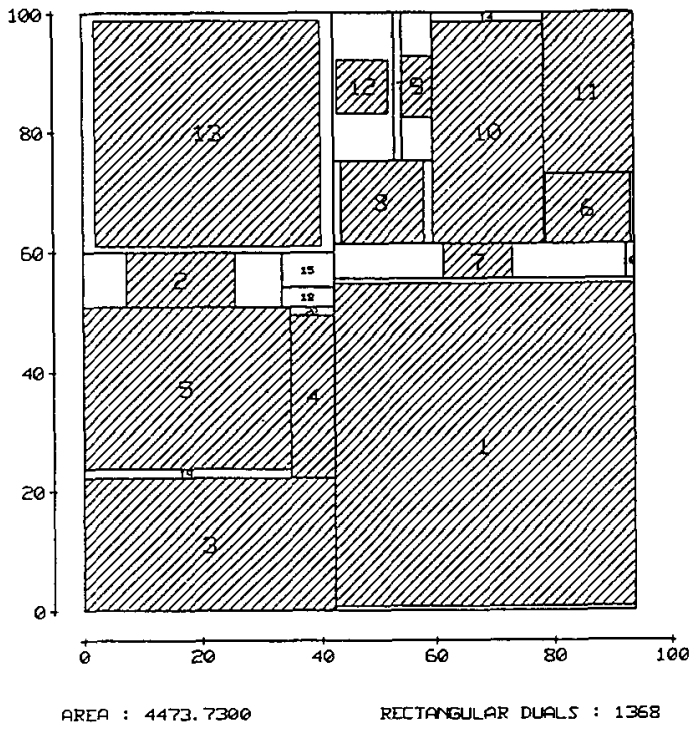


Fig. 7. A floorplan obtained by our system.

References

1. K. Maling, S. H. Mueller and W. R. Heller, "On finding most optimal rectangular package plans", *Proc. 19th DA Conf.*, (1982) pp. 663–670.
2. K. Kozminski and E. Kinnen, "Rectangular duals of planar graphs", *Networks*, **15(2)** (1985) pp. 145–157.
3. K. Koike, M. H. Yoo, K. Tani, S. Tsukiyama and I. Shirakawa, "A chip floor-plan technique with the use of rectangular dual graph", *Mono. IECE of Japan*, **CAS85-144** (1986) (in Japanese).
4. J. Bhasker and S. Sahni, "A linear time algorithm to check for the existence of a rectangular dual of a planar triangulated graph", *Networks*, **17(3)** (1987) pp. 307–317.
5. Y. T. Lai and S. M. Leinwand, "Algorithms for floorplan design via rectangular dualization", *IEEE Trans. CAD/ICAS*, **CAD-7(12)** (1988) pp. 1278–1289.
6. K. Ueda, H. Kitazawa and I. Harada, "CHAMP: Chip floor plan for hierarchical VLSI layout design", *IEEE Trans. CAD/ICAS*, **CAD-4(1)** (1985) pp. 12–22.
7. U. P. Lauther, "A min-cut placement algorithm for general cell assemblies based on a graph representation", *Proc. 16th DA Conf.* (1979) pp. 1–10.
8. O. Ore, *The Four-Color Problem*, Academic Press, New York, 1967.
9. S. Tsukiyama, K. Koike and I. Shirakawa, "An algorithm to eliminate all complex triangles in a maximal planar graph for use in VLSI floor-plan", *Proc. ISCAS'86*, (1986) pp. 321–324.
10. Y. Sun and K. H. Yeap, "Edge covering of complex triangles in rectangular dual floorplanning", private communication, 1992.
11. F. Harary, *Graph Theory*, Addison-Wesley Publishing Co., Reading, 1969.

12. K. Tani, S. Tsukiyama and I. Shirakawa, "A floor-plan system using a rectangular dual", *Proc. Joint Tech. Conf. on Circuits/Systems, Computers, and Communications (JTC-CSCC'88)*, (Seoul, 1988) pp. 310–315.
13. S. Wimer, I. Koren and I. Cederbaum, "Floorplans, planar graphs, and layouts", *IEEE Trans. CAS*, **CAS-35(3)** (1988) pp. 267–278.
14. L. Stockmeyer, "Optimal orientations of cells in slicing floorplan designs", *Information and Control*, **57(2/3)** (1983) pp. 91–101.
15. R. H. J. M. Otten, "Efficient floorplan optimization", *Proc. ICCD'83*, (1983) pp. 499–502.
16. K. Tani, S. Tsukiyama and I. Shirakawa, "Area-efficient drawings of rectangular duals for VLSI floor-plan", *Proc. ISCAS'88*, (1988) pp. 1545–1548.
17. M. Jabri, "Automatic building of graphs for rectangular dualization", *Proc. 25th DA Conf.*, (1988) pp. 638–641.
18. W. M. Dai and E. S. Kuh, "Simultaneous floor planning and global routing for hierarchical building-block layout", *IEEE Trans. CAD/ICAS*, **CAD-6(5)** (1987) pp. 828–837.
19. J. Hopcroft and R. Tarjan, "Efficient planarity testing", *J.ACM*, **21(4)** (1974) pp. 549–568.
20. M. Burstein, "Analysis of a network partitioning technique", *Proc. ISCAS'82*, (1982) pp. 477–480.
21. J. Hopcroft and R. Tarjan, "Dividing a graph into triconnected components", *SIAM J. Comput.* **2(3)** (1973) pp. 135–158.
22. V. Chvatal, "A greedy heuristic for the set-covering problem", *Mathematics of Operations Research*, **4(3)** (1979) pp. 233–235.
23. G. N. Frederickson, J. Ja'ja', "Approximation algorithms for several graph augmentation problems", *SIAM J. Comput.* **10(2)** (1981) pp. 270–283.

Algorithmic Aspects of VLSI Layout
Eds. M. Sarrafzadeh and D. T. Lee
© World Scientific Publishing Co.

CONSTRAINED VIA MINIMIZATION AND SIGNED HYPERGRAPH PARTITIONING

CHUAN-JIN SHI
Department of Computer Science, University of Waterloo
Waterloo, Ontario, Canada N2L 3G1

ABSTRACT

The constrained via minimization (CVM) problem of two-layer routing is the problem of assigning wire segments to two layers so that the number of vias is minimized. A special case of the CVM problem — with no more than three-way splits — has a nice graph-theoretic formulation and can be solved in polynomial time. However, only *ad hoc* formulations were known for the general CVM problem.

We have developed a new approach to constrained via minimization. We use a novel graph notion, called *signed hypergraphs*, to represent routings containing multiple-way splits. Signed hypergraphs are a natural extension of graphs by allowing one edge to connect more than two vertices, and one edge incident to vertices in two possible ways marked by the signs "+" and "−", respectively. An edge is balanced by a bipartition if all its vertices marked with "+" belong to one group and all its vertices marked with "−" belong to the other group. The formulation of the general CVM using this notion turns out to be surprisingly simple and general: it is a new combinatorial problem of finding such a bipartition that maximizes the number of balanced edges in a signed hypergraph. This problem is referred to as the maximum balance problem.

In this paper, we describe a linear-time heuristic for solving the maximum balance problem. We also present a technique for improving the solution quality using the property of signed hypergraphs. Experimental results with the CVM problem indicate that the algorithm generates optimal or near optimal solutions for all the test benchmark examples.

Keywords: VLSI layout, via minimization, signed hypergraph, local search.

1. Introduction

In the design of integrated circuits (ICs) and printed circuit boards (PCBs), vias are used to provide electrical connection between wires in different layers. It is desirable to minimize the number of vias, since vias usually cause the decrease of circuit yield and reliability, the increase of PCB manufacturing cost, and the use of more chip area. In this paper we study two-layer *constrained via minimization* (CVM) problem. Given a *partial routing* consisting of a set of wire segments after the placement of circuit modules and the routing of signal nets, the problem is to assign each segment to one of the two available layers in such a way that the number of vias required is minimized.

The CVM problem was first addressed by Hashimoto and Stevens[1] and has been extensively studied in the past two decades.[2,3,4,5,6,7,8,9,10,11,12,13,14] For a special case of the problem with $k \leq 3$, where k is the number of wire segments *split from* a single connection point, nice theoretical results have been obtained by a series of studies by Chen, Kajitani and Chan,[4] Pinter,[12] Kuo, Chern and Shing[8] and Barahona.[2] It is now clear that this case of the problem can be formulated as the maximum cut problem in a planar graph and an optimal solution can be found by $O(n^{1.5} \log n)$ algorithms, where n is the number of wire segments, more precisely, the number of clusters (Refer to Sec. 3 for the definition of clusters).

Despite of these results, few industrial computer-aided systems incorporate CVM into practical uses. This is partially due to two reasons. First, those polynomial CVM algorithms involve complicated graph manipulations. They require a fair amount of CPU running time and are rather difficult to implement. Second, those algorithms are only applicable to the case with $k \leq 3$. Although this case dominates routings generated by the current routing methods, a robust computer-aided design tool cannot rely on this phenomenon. As a matter of fact, in the case of grid-based layouts, as many as eight segments may be split from a single point. Moreover, in order to obtain globally optimum solutions, it is desirable to merge adjacent split points into a special split point, called the *via candidate zone*, which can have arbitrarily large degree.[4]

Research towards the general CVM problem is divided in two directions. One direction is to model a multi-way split by weighted two-way splits.[12,13,14] There are two weighting schemes proposed, which either *underestimate*[13] or *overestimate*[14] multi-way splits. Recent study indicates that it is unlikely that there exists an exact weighting scheme.[15] In addition, both schemes may lead to a nonplanar graph representation. The other direction is to represent a multi-way split by certain kinds of graph structures. This includes the effort of Chen, Kajitani and Chan,[4] Naclerio, Masuda and Nakajima,[10] and Chang and Du.[3] However, graph structure representations are rather *ad hoc*, and thus it is difficult to derive efficient solutions with such formulations.

We have developed a new approach to two-layer constrained via minimization. We introduce a novel graph notation, called *signed hypergraphs*, to characterize the general CVM problem. Signed hypergraphs are a natural extension of graphs by allowing that an edge may connect more than two vertices, and an edge connects to a vertex in two possible ways marked by the signs "+" and "−" respectively. An edge is balanced by a bipartition if all its vertices marked with "+" belong to one group and all its vertices marked with "−" belong to the other group. The formulation of the general CVM using this notion turns out to be surprisingly simple and general: it is a new combinatorial problem of finding the bipartition that maximizes the number of balanced edges in a signed hypergraph. This problem is referred to as the *maximum balance* problem.

In this paper, we describe a linear-time heuristic for solving the maximum balance problem. It provides not only a simple and fast solution for the CVM problem,

but is also applicable to a variety of problems that can be formulated as the maximum balance problem. We present a technique to improve the solution quality using some properties of signed hypergraphs.

The paper is organized as follows. Section 2 introduces basic notation. Section 3 defines the CVM problem. Section 4 reduces the CVM problem to the maximum balance problem in a signed hypergraph. A simple and fast heuristic algorithm for solving the maximum balance problem is described in Sec. 5. An improvement technique is presented in Sec. 6. Experimental results are described in Sec. 7. Section 8 concludes the paper.

2. The Notion of Signed Hypergraphs

Let $V = \{v_i | i = 1, \ldots, n\}$ be a set of elements called *vertices*, and let $E = \{(e_j^+, e_j^-) | j = 1, \ldots, m\}$ be a family of unordered pairs of subsets of V, where e_j^+ and e_j^- are subsets of V. For convenience, we use e_j to denote (e_j^+, e_j^-). The pair $H = (V, E)$ is said to be a *signed hypergraph*, or briefly an *s-hypergraph*, if (1) $e_j^+ \bigcap e_j^- = \emptyset, j = 1, \ldots, m$, (2) $e_j^+ \bigcup e_j^- \neq \emptyset, j = 1, \ldots, m$, and (3) $\bigcup_{j=1}^{m} (e_j^+ \cup e_j^-) = V$. Then e_j is called an *edge* of H, and e_j^+ and e_j^- are two *blocks* of the edge. All the vertices in e_j are said to be *incident* with the edge, *adjacent* to each other, and are called *end-vertices* of the edge. To distinguish vertices in the two blocks of e_j, we call those in e_j^+ *positive end-vertices* and those in e_j^- *negative end-vertices*. The number of vertices in e_j is the *degree* of e_j, denoted as $|e_j|$. We further exclude edges with degree smaller than two. With a slight abuse of notation, we use v_i to denote the set of edges that contain v_i if no danger of ambiguity. The number of edges in set v_i is the *degree* of v_i, denoted as $|v_i|$. We denote $max\{|v_i|, i = 1, \ldots, n\}$ by l. The *size* of a signed hypergraph is defined as $p = \sum_{j=1}^{m} |e_j|$. Note that $p = \sum_{i=1}^{n} |v_i|$ also holds.

A pictorial representation of a signed hypergraph is shown in Fig. 2(a). It represents the signed hypergraph defined by the pair (V, E) where $V = \{v_1, \ldots, v_4\}$, $E = \{e_1, \ldots, e_6\}$, $e_1 = (\{v_4\}, \{v_1\})$, $e_2 = (\{v_4\}, \{v_1\})$, $e_3 = (\{v_4\}, \{v_1\})$, $e_4 = (\{v_3, v_4\}, \{v_2\})$, $e_5 = (\{v_2, v_4\}, \emptyset)$, and $e_6 = (\{v_2, v_4\}, \emptyset)$. An edge is presented by an open curve with possibly multiple end-points, similar to an electrical net, instead of a circle used in hypergraph theory.[16] Two signs "+" and "−" are used to distinguish vertices in the two blocks of an edge.

Remark 1. In order to gain more insight into the notion of signed hypergraphs, we may define the *incidence matrix* $Q = (q_{ij})$ as follows:

$$q_{ij} = \begin{cases} +1, & \text{if vertex } v_i \text{ is a positive end-vertex of edge } e_j \\ -1, & \text{if vertex } v_i \text{ is a negative end-vertex of edge } e_j \\ 0, & \text{otherwise .} \end{cases}$$

Then signed hypergraphs are a generalization of graphs by allowing (1) more than two nonzero entries in each column of the incidence matrix, and (2) an entry to take on the value -1 as well as 0 or 1.

A *bipartition* π of V is a separation of V into two disjoint *groups* V^+ and V^-. For convenience, we say that V^+ is the *positive group* and V^- the *negative group*. An edge is said to be *balanced* by a bipartition if all its positive end-vertices belong to one group, and all its negative end-vertices belong to the other group. The *maximum balance problem* is to find such a bipartition that maximizes the number of balanced edges.

3. Two-Layer Constrained Via Minimization

It is assumed that the placement of circuit modules and the routing of signal nets have already taken place. It is also assumed that only two layers are available for routing. A *signal net* is a wire that electrically connects a specified set of terminals. Wires of different signal nets that cross each other, overlay, or otherwise violate physical design rules if placed on the same layer, are called *conflicting*, and should be assigned to different layers. The switch of the wire of a signal net from one layer to the other is physically accomplished by a *via*.

Due to the restriction of physical design rules, not all the positions in a wire can accommodate a via. We assume that we are only allowed to place vias at certain pre-specified positions, called *potential vias*. A wire is separated into *wire segments* by potential vias.

Remark 2. A wire segment defined above is not necessarily a single horizontal or vertical line segment as in most CVM studies. For example, if we do not allow placement of a via at a multi-way split point, such a multi-way split is regarded as one wire segment according to our definition.

We start with a *(partial) routing*, denoted by R, which is a collection of wire segments in all the signal nets — without specifying which layer the wire segments belong to — and a set of potential vias. We would like to find a *feasible* layer assignment, i.e., no two conflicting wire segments are assigned the same layer. The *constrained via minimization* (CVM) problem is to find such a feasible layer assignment of wire segments that minimizes the number of vias required.

Remark 3. Usually, CVM is used as a post-processor to existing routers. In this case, potential vias can be selected as vias that appear in the given physical routing generated from automatic or manual routing techniques, or selected according to some rules such as outlined in Ref. 7.

Consider Fig. 1(a) where there are five two-terminal signal nets, N_a, N_b, N_c, N_e, N_f and one three-terminal net N_d. There are six potential vias, marked by o. There are eight cross-overs of wire segments, and one knock-knee connection, marked by a $\square$. Both knock-knee points and cross-overs represent a conflict of two wire segments.

The entire routing in Fig. 1(a) is partitioned by six potential vias into four subroutings called wire-segment clusters, as shown in Fig. 1(b). In order to define wire-segment clusters precisely, we construct a so-called *constraint graph* for a partial routing as follows: each wire segment is represented by a vertex, and there exists an edge between two vertices if and only if there exists a conflict between the corresponding two wire segments. A (*wire-segment*) *cluster* is a maximal connected component in the constraint graph.[16,17] A cluster is said to be *incident* with a potential via, if there exists a wire segment in the cluster that meets the potential via.

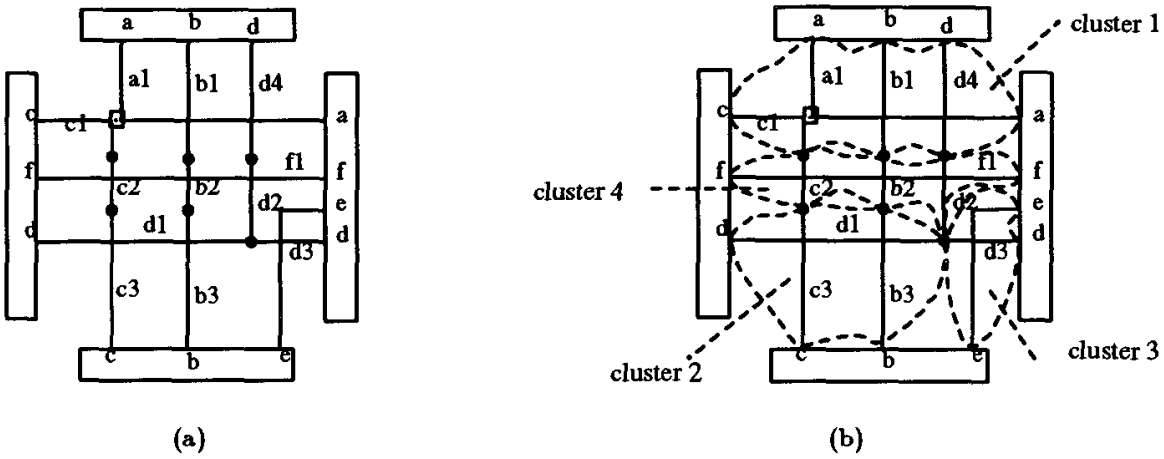


Fig. 1. A physical routing R_1 (a) and its clusters (b).

We further assume that there always exists a feasible layer assignment for a given partial routing. This is true when CVM is used as an improvement process, i.e., the routing already has a valid layer assignment and the original vias are used as potential vias. It is also true when the routing is done under the Manhattan model and potential vias are selected by some rules.[7]

Remark 4. If we do not know the feasibility of layer assignment for a given partial routing, we need to consider both the feasibility and the via minimization. This situation is not dealt with here and the interested readers may refer to Ref. 17.

4. Formulation of Constrained Via Minimization

In this section we show how the CVM problem can be formulated as a signed hypergraph problem. A basic observation that simplifies the formulation is as follows. If there exists a feasible layer assignment for the entire routing, then wire segments in each cluster can be assigned to the two layers so that no two conflicting wire segments are in the same layer. In other words, all the wire segments in a cluster are uniquely partitioned to two sets so that each set of wire segments must all be assigned to one layer. Therefore, all the wire segments in a cluster should be

treated as a whole. We arbitrarily choose one set of wire segments as *reference wire segments*, the other set of wire segments as *non-reference wire segments*.

The primary objects of CVM are a set of clusters and a set of potential vias. Incident to each potential via are a subset of clusters, where each cluster may be incident to the potential via through its reference wire segment or its non-reference wire segment. It is natural to capture the above objects by a signed hypergraph. The derivation of such a signed hypergraph involves the following steps:

1. Represent each cluster by a vertex, denote the set of vertices by V.
2. Represent each potential via by an edge, denote the set of edges by E.
3. Establish the incidence of edges and vertices as follows: If a cluster is incident to a potential via through a reference (non-reference, respectively) wire segment, then the corresponding vertex is a positive (negative, respectively) end-vertex of the corresponding edge.

Consider the routing in Fig. 1. Clusters 1, 2, 3, and 4 are represented by vertices v_1, v_2, v_3 and v_4, respectively. Wire segments $\{a1\}$, $\{c2, b2, d2\}$, $\{c3, b3\}$ and $\{d3\}$ are chosen as reference wire segments. The resulting signed hypergraph is shown in Fig. 2(a).

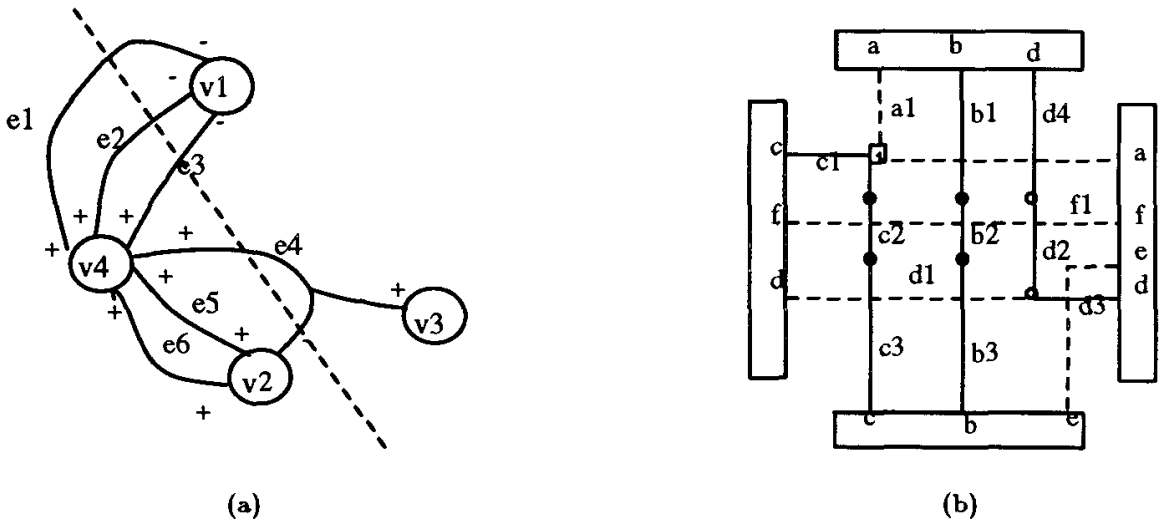
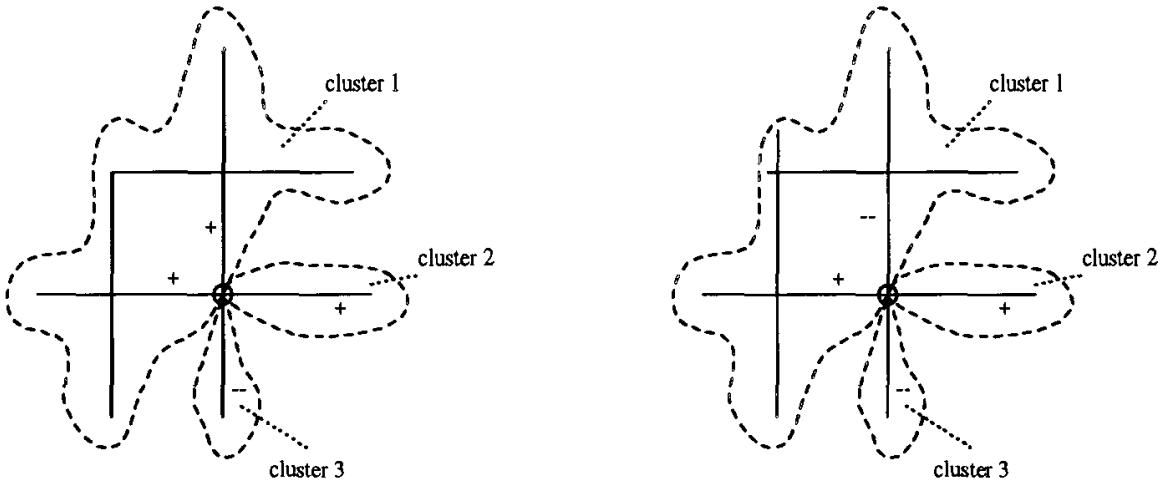


Fig. 2. Signed hypergraph of R_1 and the CVM solution of R_1.

By such a transformation, assigning wire segments to the two layers corresponds to partitioning vertices into two groups. The condition for not requiring a real via at the position of a potential via is that all the wire segments joined at that potential via are assigned to the same layer. That is, all the positive vertices of the corresponding edge are in one group of a bipartition and all the negative end-vertices the other group. This is the concept of balance. Therefore, layer assignment aimed at via minimization amounts to partitioning all the vertices into two groups so as to maximize the number of balanced edges.

The pair (V, E) obtained directly from the above transformation is not a signed hypergraph in a strict sense. A potential via may be represented by an edge with one vertex appearing more than once in a block of the edge or with one vertex appearing in both blocks, which violates the first two conditions of signed hypergraphs. One such example is illustrated in Fig. 3. There are three clusters incident to one potential via. Wire segments marked by "+" are chosen as reference wire segments. This leads to edge $(\{1,1,2\},\{3\})$ for case (a) and edge $(\{1,2\},\{1,3\})$ for case (b). Fortunately, we can do a simple modification such that all the resulting edges satisfy the two conditions of signed hypergraphs without affecting the solution. In the former case, we simply replace all the appearances of a vertex by a single one. In the latter case, the edge will never be balanced. This is called an *essential via*[1,3]: a via is always required at the position corresponding to the edge. Therefore, when the latter case happens, we report an essential via and delete the edge. In addition, an edge incident on only one vertex can be simply removed. Now the modified pair $H = (V, E)$ turns out to be exactly a signed hypergraph.



(a) edge ({1,1,2}, {3})

(b) edge ({1,2}, {1,3})

Fig. 3. Two cases of a non-standard edge.

We have established the following result:

Theorem 1. *The CVM problem for routing R is equivalent to the maximum balance problem in signed hypergraph H of R.*

Consider the signed hypergraph in Fig. 2(a). Bipartition $(\{v_1, v_3\}, \{v_2, v_4\})$ is an optimal solution of the maximum balance problem, where the number of unbalanced edges is one. This corresponds to an optimal layer assignment shown in Fig. 2(b), which requires one via.

5. An Iterative Improvement Heuristic

In this section, we describe an iterative improvement heuristic for solving the maximum separation problem in a signed hypergraph. The heuristic is developed much along the lines of the work of Fiduccia and Mattheyses on netlist (hypergraph) partitioning.[18]

5.1. The Basic Idea

Let π be a bipartition of V in H, $\mathcal{F}(\pi)$ be the *cost* of π, which is the number of edges that are not sign-separated by π. For each vertex v_i in V, we define *gain* $g(i)$ as the reduction of the cost if vertex v_i is moved from its current group of π to its complementary group. We use g to denote $g(1), g(2), \ldots, g(n)$.

The basic idea is to start with an arbitrary initial bipartition, and repeatly to *move* one vertex each time from its current group to its complementary group. Each time, a vertex of maximum gain is chosen to move, and it is immediately *locked* after

Vertex pushing algorithm

```
01  Input: H — a signed hypergraph;
02  Output: π_output — the found partition;
03  push_vertex(H, π_output)
04    {
05    π_best = get_initial_partition();
06    F_best = cost_of_partition(π_current);
07
08    do      /* one pass */
09      {
10      π_current = π_last_best = π_best;
11      F_current = F_last_best = F_best;
12      compute_gain(H, π_current, g);
13      set all the vertices unlocked;
14      for (v_b : unlocked vertex of maximum gain MAXGAIN)
15        {
16        π_current(b) = the complementary of π_current(b);
17        lock v_b;
18        F_current = F_current + MAXGAIN;
19        update_gain(H, v_b, g);
20        if (F_current < F_best)
21          {
22          F_best = F_current;
23          π_best = π_current;
24          }
25        }
26      } while (F_best < F_last_best)
27    π_output = π_last_best;
28    }
```

Fig. 4. The vertex pushing algorithm.

the move, i.e., any further move of this vertex is not allowed. We say that a *pass* of the algorithm is a sequence of successive moves until all the vertices become locked.

Figure 4 shows the pseudocode of an algorithm, called *push_vertex*, for solving the maximum balance problem, which embodies the above idea. Line 05 establishes an arbitrary initial bipartition. One pass of the algorithm (lines 10 to 25) starts with that initial bipartition as $\pi_{current}$ and first invokes *compute_gain* to calculate g for $\pi_{current}$ (line 12). A vertex of maximum gain in g is then chosen and moved from its current group to its complementary group. The cost and the gains are adjusted accordingly. This process continues with another vertex of new maximum gain until all the vertices of V become locked. With the bipartition of minimum cost found in the pass as an initial bipartition, we start another pass of the algorithm in an attempt to reduce further the cost of bipartition. Note that whenever there is a bipartition such that $g(i) \leq 0$ for all the vertices, we reach a local optimum, i.e., it is impossible to reduce further the cost by any single move. However, we still continue the move with the expectation to climb out of local minima.

5.2. Gain Computation

Procedure *compute_gain* calculates g, the gains of all the vertices in H for a given bipartition. In order to derive *compute_gain*, let us consider $g(i)$ of vertex v_i. If we denote the contribution of edge e_j to gain $g(i)$ by $g(j,i)$, then we have:

$$g(i) = \sum_{j=1}^{m} g(j,i)$$

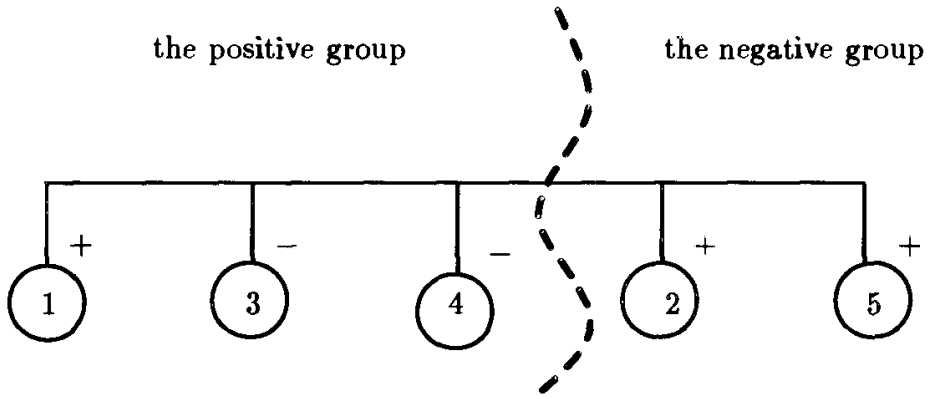
Clearly $g(j,i)$ has three possible values defined below:

$$g(j,i) = \begin{cases} 1, & \text{when } v_i \text{ is moved, } e_j \text{ changes from unbalanced to balanced} \\ -1, & \text{when } v_i \text{ is moved, } e_j \text{ changes from balanced to unbalanced} \\ 0, & \text{otherwise .} \end{cases}$$

To determine $g(j,i)$, we observe that the move of vertex v_i has the same effect on the separation status of edge e_j as the move of any other vertex that is in the same group and with the same sign as v_i, or in the different group and with the different sign as v_i. The number of such vertices is called *the critical number* of vertex v_i and denoted as $\mathcal{N}_i(e_j)$. (See Fig. 5 for an example.) Symmetrically, $\overline{\mathcal{N}_i(e_j)}$ is used to denote the number of vertices that either belong to the same group as v_i but with the different sign, or belong to the different group as v_i but with the same sign. The following propositions follow immediately from the definitions.

Proposition 1. $\mathcal{N}_i(e_j) + \overline{\mathcal{N}_i(e_j)} = |e_j|$.

Proposition 2. *An edge e_j is balanced iff $\mathcal{N}_i(e_j) = |e_j|$ for all vertices v_i in e_j.*



$$\mathcal{N}_1 = 1 = \mathcal{N}_+$$
$$\mathcal{N}_3 = \mathcal{N}_4 = \mathcal{N}_2 = \mathcal{N}_5 = 4 = \mathcal{N}_-$$

Fig. 5. Calculation of critical numbers.

Proposition 3. $g(j,i) = -1$ *iff* $\mathcal{N}_i(e_j) = |e_j|$.

Proposition 4. $g(j,i) = 1$ *iff* $\mathcal{N}_i(e_j) = 1$.

We give a proof of Proposition 4 as below.

Proof. If $g(j,i) = 1$, that means, moving v_i from its current group to the complementary group will enable e_j to become balanced when it was not balanced, then by Proposition 2, $\mathcal{N}_i(e_j) = |e_j|$ after the move. So $\overline{\mathcal{N}_i(e_j)} = |e_j| - 1$ before the move. Hence $\mathcal{N}_i(e_j) = 1$ follows from Proposition 1.

If $\mathcal{N}_i(e_j) = 1$, since $|e_j| > 1$ from the definition of signed hypergraphs, then $\mathcal{N}_i(e_j) \neq |e_j|$. Hence by Proposition 2, e_j is not balanced. If we move vertex v_i, then $\mathcal{N}_i(e_j)(after) = 1 + \overline{\mathcal{N}_i(e_j)}(before) = \mathcal{N}_i(e_j)(before) + \overline{\mathcal{N}_i(e_j)}(before) = |e_j|$. Therefore e_j is balanced after the move. Hence $g(j,i) = 1$. □

Figure 6 shows the pseudocode of *compute_gain*. It first computes the critical numbers for all the edges, then calculates the gains based on the edge's critical numbers. Note that all the positive (negative, respectively) vertices in the positive group and all the negative (positive, respectively) vertices in the negative group have the same critical number for a given bipartition. We denote it by $\mathcal{N}_+(e_j)$ ($\mathcal{N}_-(e_j)$, respectively). The critical number of each vertex is either $\mathcal{N}_+(e_j)$ or $\mathcal{N}_-(e_j)$, dependent upon its sign and its group. The validity of this procedure has been established in Proposition 3 and 4.

Initial Gain Computation

```
01  Input: H — a signed hypergraph;
02         π — a partition;
03  Output: g — the gains of all the vertices.
04  compute_gain(H, π, g)
05    {
06    /* calculate critical numbers */
07    for (e_j : each edge in H)
08      {
09      N+(e_j) = 0;
10      N−(e_j) = 0;
11      for (v_i : each vertex in e_j)
12        {
13        if (v_i has the positive sign in the positive group
14              or has the negative sign in the negative group)
15          N+(e_j) = N+(e_j) + 1;
16        else
17          N−(e_j) = N−(e_j) + 1;
18        }
19      }
20    /* calculate gains */
21    for (v_i : each vertex in H)
22      {
23      g(i) = 0;
24      for (e_j : each edge of H containing vertex v_i)
25        {
26        if (v_i has the positive sign in the positive group
27              or has the negative sign in the negative group)
28          Ni(e_j) = N+(e_j);
29        else
30          Ni(e_j) = N−(e_j);
31        if (Ni(e_j) == 1)    /* g(j,i) = 1 */
32          g(i) = g(i) + 1;
33        else if (Ni(e_j) == |e_j|)    /* g(j,i) = −1 */
34          g(i) = g(i) − 1;
35        }
36      }
37    }
```

Fig. 6. The gain initialization algorithm.

5.3. Gain Updating

Figure 7 describes procedure *update_gain* for calculating g after the move of vertex v_b. It makes use of the known g before the move, and updates only those portions that may be affected by the move.

Proposition 5. *Procedure update_gain computes the gains of all the vertices after the move of vertex v_b correctly.*

Gain Maintenance

```
01  Input: H — a signed hypergraph;
02         v_b — the base vertex;
03  Output: the updated g.
04  update_gain(H, v_b, g)
05    {
06    tmp: a local variable;
07    for (e_j : each edge in H containing v_b)
08      {
09      /* before the move */
10      if (N_b(e_j) == |e_j|)
11        increment gains of all unlocked vertices on e_j;
12      else if (N_b(e_j) == |e_j| - 1)
13        decrement the gain of the only vertex
14          that has the critical number 1 if it is unlocked;
15      /* adjust the edge's critical numbers */
16      if (v_b has the positive sign in the positive group
17            or the negative sign in the negative group)
18        tmp = 1;
19      else
20        tmp = -1;
21      N+(e_j) = N+(e_j) + tmp;
22      N-(e_j) = N-(e_j) - tmp;
23      /* after the move */
24      if (N_b(e_j) == |e_j|)
25        decrement gains of unlocked vertices on e_j;
26      else if (N_b(e_j) == |e_j| - 1)
27        decrement the gain of the only vertex
28          that has the critical number 1 if it is unlocked;
29      }
30    }
```

Fig. 7. The gain maintenance algorithm.

Proof. Let us consider $g(i)$ where $i \neq b$. (We do not need to consider $g(b)$, since v_b is locked after the move.) Only the gain contribution $g(j,i)$ of an edge e_j in set v_b — the set of edges that contain vertex v_b — to $g(i)$ may be affected by the move of v_b. Therefore $g(i)$ after the move is equal to the original $g(i)$ before the move plus $g(j,i)$ after the move for all e_j in set v_b and minus $g(j,i)$ before the move for all the edges e_j in set v_b. From Proposition 3, $g(j,i) = -1$ iff $\mathcal{N}_i(e_j) = |e_j|$. That is, $\mathcal{N}_b(e_j) = |e_j|$. From Proposition 4, $g(j,i) = 1$ iff $\mathcal{N}_i(e_j) = 1$. That is $\mathcal{N}_i(e_b) = |e_j| - 1$, since $i \neq b$. Therefore procedure *update_gain* correctly calculates the gains for all the vertices after the move. □

5.4. Data Structures

In the algorithms described above, the following primitive operations have been used:

- OP_1 : Given a vertex v_i, return all the edges that contain v_i.

- OP_2 : Given an edge e_j, return all the vertices contained in e_j.
- OP_3 : Return a vertex of highest gain.

We maintain two adjacent-list representations for a signed hypergraph. One consists of an array of n lists, one for each vertex v_i and containing a linked-list of pointers to all the edges that contain v_i. The other consists of an array of m records, one for each edge e_j and containing a pair of integers describing $\mathcal{N}_+(e_j)$ and $\mathcal{N}_-(e_j)$, and a linked-list of pairs (*pointer*, *sign*), where each *pointer* points to a vertex contained in e_j with the indicated *sign*. Either of the two representations can be generated from the other in $O(p)$ time.

Proposition 6. *Using the above two adjacent-list representations of a signed hypergraph, OP_1 and OP_2 are supported in $O(|v_i|)$ time and $O(|e_j|)$ time, respectively.*

We apply bucket sort to support OP_3. Since the gain of moving one vertex is at most l and at least $-l$, we maintain an array $BUCKET[-l, \ldots, l]$ to keep the gains of all the vertices, that is g; $BUCKET[i]$ contains a doubly-linked list of all the vertices with gain currently equal to i. We also maintain for each vertex a direct pointer to its link in *BUCKET*. In addition, a *MAXGAIN* index is used to keep track of the bucket having a vertex of highest gain. This index is updated by decrementing it whenever its bucket is found to be empty and resetting it to a higher bucket whenever a vertex moves to a bucket above *MAXGAIN*. Whenever a vertex is locked, it is removed from *BUCKET*.

In order to maintain *BUCKET*, we need the following primitives:

- OP_4 : Given a vertex and its gain, insert it in *BUCKET*.
- OP_5 : Given a vertex, delete its gain from *BUCKET*.
- OP_6 : Given a vertex, modify its gain in *BUCKET*.

Proposition 7. *Using the above bucket data structures, OP_4, OP_5 and OP_6 are supported in constant time.*

5.5. Complexity Analysis

The time complexity of algorithm *push_vertex* depends on the number of passes. Our experiments have shown that it typically takes only a few passes to converge. In the following we study the time complexity of one pass of the algorithm.

Proposition 8. *The time complexity of compute_gain is $O(p)$.*

Proof. Immediate by counting the number of times each statement executes. □

In order to analyze the computational cost of *update_gain* and the maintenance of the bucket data structure, we need to examine how many times statements in lines

11, 13 and 14, 25, 27 and 28 in Fig. 7 are performed during one pass of algorithm *push_vertex*. Each of those four statements takes $O(|e_j|)$ time in the worst case. For convenience, we say that each performs *one update operation*.

Proposition 9. *Only a constant number of update operations are required for each edge during one pass of algorithm push_vertex.*

Proof. An update operation is required for edge e_j iff $\mathcal{N}_b(e_j) = |e_j|$ or $\mathcal{N}_b(e_j) = |e_j| - 1$ for the move of some $v_b \in e_j$ (lines 10, 12, 24 and 26 in *update_gain*). Since $\mathcal{N}_b(e_j)$ is equal to either $\mathcal{N}_+(e_j)$ or $\mathcal{N}_-(e_j)$, suppose $\mathcal{N}_b(e_j) = \mathcal{N}_+(e_j)$, $\mathcal{N}_+(e_j)$ could be equal to $|e_j|$ at most twice — in two succeeding moves, one after the first and the other before the second move. This is because any vertex moved is locked after the move, hence $\mathcal{N}_+(e_j) \leq |e_j| - 1$ in the rest of the pass. For the same reason, each of $\mathcal{N}_+(e_j) = |e_j| - 1$, $\mathcal{N}_-(e_j) = |e_j|$, and $\mathcal{N}_-(e_j) = |e_j| - 1$ could be satisfied at most twice. Therefore, the number of required update operations is no more than eight for each edge during one pass of the algorithm. □

Remark 5. If we distinguish locked vertices and unlocked vertices for counting the critical numbers, then we could show that three update operations are sufficient for each edge.

Proposition 10. *Operation OP_3 and the maintenance of the bucket data structures to support OP_3 take $O(p)$ time during one pass of algorithm push_vertex.*

Proof. Operation OP_3 is performed n times during one pass. Each time the access of *MAXGAIN* to obtain the link of a vertex of maximum gain in *BUCKET* takes constant time. So we need to examine the total cost of maintaining *MAXGAIN* and *BUCKET*.

The construction of *BUCKET* (using OP_4) for initial gains and the initialization of *MAXGAIN* in procedure *compute_gain* takes $O(n)$ time. Whenever we move a vertex, we need to delete it from *BUCKET* (OP_5). Since there are n moves, the total time spent by OP_5 is $O(n)$. If the vertex removed by OP_5 is the only one in that bucket, we need to decrement index *MAXGAIN* until we get the next non-empty bucket. In the worst case, the number of empty buckets is $2l$. Therefore $O(p)$ time is sufficient to maintain *MAXGAIN* due to all the OP_5 operations. By Proposition 4, each edge is updated a constant number of times, say c, and each update takes $O(|e_j|)$ OP_6 operations. So the total number of OP_6 operations is $\sum_{j=1}^{m} c|e_j| = O(p)$. Each time an edge is updated, the gain of any vertex on that edge can be incremented or decremented once. So there is at most one bucket that may be found to be empty during the maintenance of *MAXGAIN* for one OP_6 operation. Therefore the total time for maintaining *MAXGAIN* and *BUCKET* is $O(p)$. □

Theorem 2. *Algorithm push_vertex requires $O(p)$ time to complete one pass.*

Proof. One pass of the algorithm is described from line 10 to line 25 in Fig. 4. From Proposition 8, *compute_gain* takes $O(p)$ time. By Proposition 10, line 14 needs $O(p)$ time in total for one pass. Line 16 to line 24 are performed n times. From Proposition 9, the total time spent by *update_gain* is $O(p)$. Therefore, the time complexity of algorithm *push_vertex* is $O(p)$. □

Remark 6. There are two primary reasons why we resort to heuristics for solving the maximum balance problem. First, the CVM problem is NP-complete.[5,11] Second, if we are interested in solving the general CVM problem possibly with fixed layer assignment, we could not rely on the polynomial solvability of the special case of the CVM problem.

6. Hierarchical Refinement for Global Minima

The solutions produced by algorithm *push_vertex* are, in general, not minimal, but locally minimal with regard to the *neighborhood*. In this section, we present a refinement technique that makes use of the properties of signed hypergraphs to improve the solution quality.

We first consider an example as illustrated in Fig. 8(a). There are five vertices and four edges. The current bipartition is $(\{4\}, \{2, 3, 6, 8\})$ as illustrated by a dashed line. The edge incident on vertices 8 and 3 is not balanced. It is easy to verify that no further move of any vertex can have a positive gain; i.e., increase the number of balanced edges. However, as illustrated in Fig. 8(d), if vertices 3 and 4 are allowed to interchange, then it will further increase the number of balanced edges by 1.

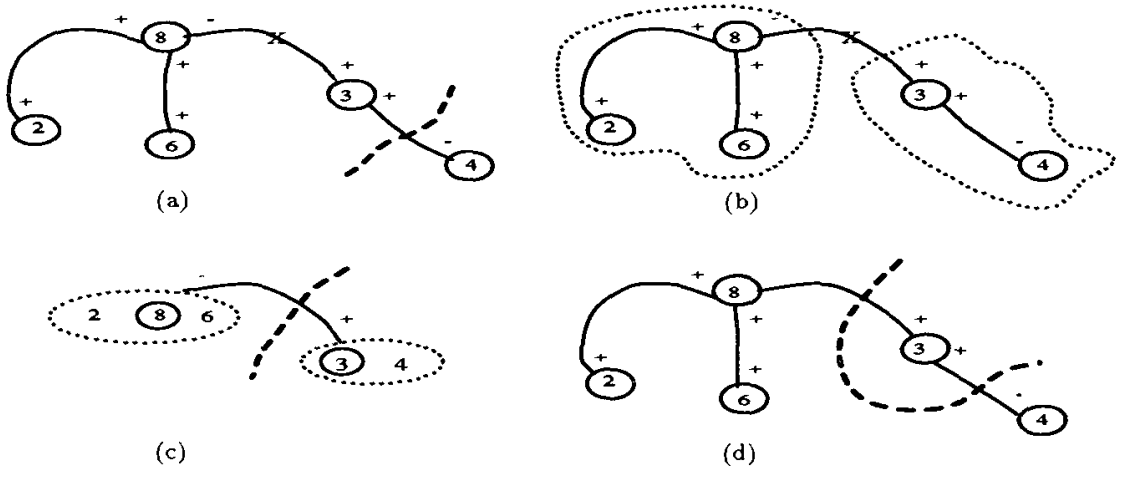


Fig. 8. Illustration of further refinement of vertex-pushing.

This example can be generalized as follows. Let $H = (V, E)$ be a signed hypergraph, and V_1 be a nonempty subset of V. Let E_1 be all the edges that are only

incident on vertices in V_1, i.e., all the end-vertices of such edges are in V_1. The pair (V_1, E_1) is a *sub-hypergraph* of H induced on V_1.

We consider one property of sub-hypergraphs. Let $\pi = (V^+, V^-)$ be a bipartition of V. All the vertices V_1 of sub-hypergraph H_1 are partitioned into the two groups by π. If we interchange all the vertices of V_1 in V^+ with all the vertices of V_1 in V^-, and denote the resulting bipartition as π', then the number of balanced edges by π' is the same as that of π. This is because all the balanced edges in E_1 remain balanced, all the unbalanced edges in E_1 remain unbalanced, and finally, all the edges not in E_1 remain as what they are.

Suppose that V is partitioned into a set of subset of vertices. For convenience, each subset is said to be a *cluster*. All the vertices in a cluster is partitioned to the two groups by π. If we interchange vertices in the two groups for a cluster, all the edges in the sub-hypergraph induced on this cluster will keep their balance status. However, edges with end-vertices in the different clusters may change their balance status. Hence we have a new optimization problem, that is to classify all the clusters of V to two categories — interchanged or not interchanged — so as to maximize the number of balanced edges. Since we do not need to consider those edges with end-vertices in a cluster, this problem can be formulated as the maximum balance problem in a reduced signed hypergraph. Let $H^r = (V^r, E^r)$ be such a reduced signed hypergraph, the construction of H^r involves the following steps:

1. Perform algorithm *push_vertex* on H, and denote the obtained bipartition by $\pi = (V^+, V^-)$.
2. Partition V into a set of clusters.
3. For each cluster, choose either those vertices in V^+ or in V^- as the reference vertices, and the other group as the non-reference vertices.
4. Represent each cluster by a vertex in V^r.
5. If an edge e in H is incident on vertices in more than one cluster, construct an edge e^r in H^r incident on the corresponding vertices in V^r. If e is incident on a reference vertex of V, then e^r is incident on the corresponding vertex in V^r with the same sign as e incident on that reference vertex in H, otherwise, e^r is incident on the corresponding vertex in V^r with the complementary sign.

A simplest way to partition V into a set of clusters is to define a maximal set of vertices that are incident with balanced edges to be a cluster. In other words, we shrink all the balanced edges and merge the vertices incident with each balanced edge to be one vertex.

Consider Fig. 8(a). All the vertices are partitioned into two clusters {2, 8, 6} and {3, 4}, as illustrated in Fig. 8(b). With vertices 8 and 3 chosen as reference vertices, the resulting reduced signed hypergraph is shown in Fig. 8(c).

Now algorithm *push_vertex* can be performed on H_r in an attempt to further increase the number of balanced edges. This process is repeated until the reduced signed hypergraph is stable, i.e., no further reduction of the cost is possible. Finally, we back trace the process to obtain the final bipartition. The reader can verify that applying algorithm *push_vertex* on H_r in Fig. 8(c) yields the final bipartition as shown in Fig. 8(d).

There are several observations that can be made on the above *hierarchical* version of algorithm *push_vertex*.

1. The algorithm takes $u \leq m$ iterations to converge, where m is the number of edges in a signed hypergraph.
2. Let $E_i \subseteq E$ be the set of balanced edges obtained by the ith run of algorithm *push_vertex*, and $|E_i|$ be the cardinality of E_i. Then $E_1 \subset E_2 \subset \ldots \subset E_{u-1} \subseteq E_u$, and thus $|E_1| < |E_2| < \ldots < |E_{u-1}| \leq |E_u|$.
3. Each iteration takes $O(p)$ time in the worst case.

7. Experimental Results

The proposed algorithm for the maximum balance problem has been implemented in the C programming language, and used as a postprocessor to reduce the number of vias for several practical routing benchmarks. The results are summarized in Table 1. These benchmarks are taken from the literature on channel routing[19,20] or on via minimization,[6] and were used by Chang and Du to evaluate their CVM algorithms.[3]

We conducted two groups of experiments, using different criteria for selecting potential vias. The first group is to choose original vias as potential vias. The results are given in the columns under "selection (1)" in Table 1, and compared with those by Chang and Du. Several observations can be made here. First, our model easily identified all the essential vias (#ev). In contrast, Chang and Du were only able to identify part of essential vias (namely EV_3) using a rather sophisticated algorithm. Second, most vias happen to be essential vias for these benchmarks. Since essential vias do not appear in our signed hypergraph formulation, the number of edges is equal to #ov minus #ev, and the number of vertices is equal to #c. The resulting signed hypergraph is very small. This is consistent with the observation of Hashimoto and Stevens.[1] Third, the number of vias after reduction by our program is the same as that of Chang and Du. These results were verified to be optimal by Chang and Du. Note that except for routing 7, the number of vias (#v) is the same as the number of essential vias (#ev). Fourth, only a few passes are required. Refinement is only needed for routing 1.

The second group of experiments is conducted by selecting a maximal set of potential vias, i.e, wherever there is a conflict of two wire segments, there are four potential vias "enclosing" this conflict. The results are shown in the columns under "selection (2)" in Table 1. We achieved further reduction of vias for routing 2 to 7. Both the number of potential vias and the number of clusters are significantly

Table 1. Experimental Results.

	selection (1)						selection (2)				
	#ov	#c	$\#EV_3$	#ev	#v	#p	#pv	#c	#v	#p	Refer
1	20	20	1	1	1	1*	26	26	1	2	Fig. 2^{20}
2	6	5	2	2	2	1	11	20	2	1	Fig. 23^{19}
3	16	8	9	9	9	1	29	13	7	1	Fig. 22^{19}
4	26	10	16	16	16	1	40	13	13	1	Fig. 17^{19}
5	57	17	35	40	40	1	351	197	35	2	Fig. 25^{20}
6	91	24	70	72	72	1	756	418	68	1	Fig. 26^{20}
7	55	42	16	17	19	3	142	115	17	2	Fig. 3^{6}

$\#ov$	number of original vias	#c	number of clusters
$\#ev$	number of essential vias	$\#v$	number of vias after reduction
$\#p$	number of passes	*	result after refinement
EV_3	part of potential vias by Chang and Du	$\#pv$	number of potential vias

larger than those of scheme (1). However, even in this extreme case, the size of the resulting problem is still smaller than the earlier CVM formulation by Ciesielski and Kinnen.[6]

We observed that a lots of potential vias chosen by scheme (2) are unnecessary, i.e., we can remove them without affecting the global optimality of the solution. As a result, the size of the resulting signed hypergraph can be reduced significantly. Therefore, a new optimization problem arises: how to choose the minimum number of potential vias so that we can still obtain the global optimum solutions. This problem affects directly both the quality and efficiency of constrained via minimization. However, it has been largely ignored in the past.

8. Conclusions

In this paper, we represented a two-layer physical routing with layer assignment to be determined by a *signed hypergraph*, and formulated the via minimization problem as a new combinatorial optimization problem, namely the maximum balance problem in a signed hypergraph.

The Fiduccia and Mattheyses heuristic on netlist partitioning has been generalized to solve the maximum balance problem. We have shown that the worst-case computational cost of one pass of the heuristic grows linearly with the size of signed hypergraph. We also presented a refinement technique that makes use of the properties of signed hypergraphs to further improve the solutions.

The algorithm has been implemented in the C programming language. Experimental results are quite encouraging. Our program generated optimal solutions for all test benchmark examples taken from the literature.

Acknowledgements

This research was supported by the Natural Sciences and Engineering Council of Canada and the Information Technology Centre of Ontario. The author is grateful to the referees, especially the second referee, for comments led to a clear presentation of the entire paper, especially Sec. 5. The author would like to thank Professors John Brzozowski, Anthony Vannelli and Jiri Vlach for many helpful discussions.

References

1. A. Hashimoto and J. Stevens, "Wire routing optimizing channel assignment within large apertures", *Proc. 8th Design Automation Workshop*, June 1971, pp. 155–169.
2. F. Barahona, "On via minimization", *IEEE Trans. Circuits and Systems*, **CAS-37** (1990) 527–530.
3. K. C. Chang and D. H.-C. Du, "Efficient algorithms for layer assignment problem", *IEEE Trans. Computer-Aided Design*, **CAD-6** (1987) 67–78.
4. R. W. Chen, Y. Kajitani and S. P. Chan, "A graph-theoretic via minimization algorithm for two-layer printed circuit boards", *IEEE Trans. Circuits and Systems*, **CAS-30** (1983) 284–299.
5. H.-A. Choi, K. Nakajima and C. S. Rim, "Graph bipartition and via minimization", *SIAM J. Disc. Math.*, **2** (1989) 38–47.
6. M. J. Ciesielski and E. Kinnen, "An optimum layer assignment for routing in ICs and PCBs.", *Proc. ACM/IEEE Design Automation Conf.*, 1981, pp. 733–737.
7. M. J. Ciesielski, "Layer assignment for VLSI interconnect delay minimization", *IEEE Trans. Computer-Aided Design*, **CAD-8** (1989) 702–707.
8. Y. S. Kuo, T. C. Chern and W. K. Shih, "Fast algorithm for optimal layer assignment", *Proc. ACM/IEEE Design Automation Conf.*, 1988, pp. 554–559.
9. P. Molitor, "Constrained via minimization for systolic arrays", *IEEE Trans. Computer-Aided Design*, **CAD-9** (1990) 537–542.
10. N. J. Naclerio, S. Masuda and K. Nakajima, "Via minimization for gridless layouts", *Proc. ACM/IEEE Design Automation Conf.*, 1987, pp. 159–165.
11. N. J. Naclerio, S. Masuda and K. Nakajima, "The via minimization problem is NP-complete", *IEEE Trans. Computers*, **C-38** (1989) 1604–1608.
12. R. Y. Pinter, "Optimal layer assignment for interconnect", *Proc. IEEE Circuits and Computers Conf.*, 1982, pp. 398–401.
13. K. R. Stevens and W. M. van Cleemput, "Global via minimization in generalized routing environment", *Proc. International Symposium on Circuits and Systems*, 1979, pp. 689–692.
14. X. M. Xiong and E. S. Kuh, "A unified approach to the via minimization problem", *IEEE Trans. Circuits and Systems*, **CAS-36** (1989) 190–204.
15. C.-J. Shi, "Modeling k-way splits of physical routing for constrained via minimization", *Proc. Canadian Conf. on VLSI*, 1991, pp. 4B.5.1–4B.5.8.
16. C. Berge, *Graphs and Hypergraphs* (North-Holland, Amsterdam, 1973).
17. C.-J. Shi, "A signed hypergraph model of constrained via minimization", *Proc. Second Great Lakes Symposium on VLSI*, 1992, pp. 159–166.

18. C. M. Fiduccia and R. M. Mattheyses, "A linear-time heuristic for improving network partitions", *Proc. ACM/IEEE Design Automation Conf.*, 1982, pp. 175–181.
19. I. S. Gopal, D. Coppersmith and C. K. Wong, "Optimal wiring of movable terminals", *IEEE Trans. Computers*, **C-32** (1983) 845–858.
20. T. Yoshimura and E. S. Kuh, "Efficient algorithms for channel routing", *IEEE Trans. Computer-Aided Design*, **CAD-1** (1982) 25–35.

Algorithmic Aspects of VLSI Layout
Eds. M. Sarrafzadeh and D. T. Lee
© World Scientific Publishing Co.

THE VIRTUAL DIMENSIONS OF A STRAIGHT LINE EMBEDDING OF A PLANE GRAPH

TOSHIHIKO TAKAHASHI
Department of Information Engineering, Niigata University
Ikarashi 2-nocho, Niigata 950-21, Japan

and

YOJI KAJITANI*
Department of Electric & Electronic Engineering, Tokyo Institute of Technology
Meguro-ku Ookayama, Tokyo 152, Japan

ABSTRACT

The virtual-height (virtual-width) of a straight line embedding of a plane graph in the xy-plane is the number of vertices with pairwise distinct y-coordinates (x-coordinates). We show that any plane graph of order n has a straight line embedding of virtual-height at most $\lfloor \frac{2n+1}{3} \rfloor$ and virtual-width at most $\lceil \frac{2n+1}{3} \rceil$ and present a polynomial time algorithm which obtains such an embedding.

Keywords: Plane graph, straight line embedding, height, width.

1. Introduction

A *plane graph* is a graph with an embedding in the plane. An embedding of a plane graph G is called a *straight line embedding* when all the edges consist of straight line segments. We denote a straight line embedding of G as G^*. It is well known that any (simple) plane graph has a straight line embedding.[1,6,7]

It is an important problem to obtain a straight line embedding within a limited space in VLSI layout. The algorithms of Schnyder[5] and Fraysseix, Pach, and Pollack[2] obtain straight line embeddings on the grid within both the limited height and width. In particular, Schnyder has shown that any plane graph has a straight line embedding on the $n-2$ by $n-2$ grid.

On the other hand, the trivial lower bound for such grids is $\lceil \frac{2n-1}{3} \rceil$ by $\lceil \frac{2n-1}{3} \rceil$. (Consider a plane graph which contains $\lfloor \frac{n}{3} \rfloor$ nested triangles as its subgraphs.)

The purpose of this paper is to prove the following theorem on straight line embeddings closely related to the above problem:

*Also Japan Advanced Institute of Science and Technology.

Theorem 1 (VIRTUAL-DIMENSION THEOREM)
Each plane graph of order n has a straight line embedding of virtual-height at most $\lfloor \frac{2n+1}{3} \rfloor$ *and virtual-width at most* $\lceil \frac{2n+1}{3} \rceil$, *where the virtual-height (virtual-width) of a straight line embedding of a plane graph in the xy-plane is the number of vertices with pairwise distinct y-coordinates (x-coordinates).*

2. Embedding Algorithm

Theorem 1 will be proved by giving an embedding algorithm obtaining a desired straight line embedding G^* for a given plane graph G. Without loss of generality, we assume that G is a *plane triangulation.*

First, we introduce two operations *contraction* and *expansion* used in the algorithm. A cycle of length three is a *separating triangle* if the finite region of the plane separated by this cycle contains at least one vertex. A separating tetragon is similarly defined. An edge $e = (u, v)$ is *contractible* if it is not a side of a separating triangle. Note that, by this definition, the outer cycle of plane triangulation G is a separating triangle and the edges of this cycle are not contractible. The following operation is defined as a *contraction of e*: shrinking contractible edge e into one vertex v (or u) on the plane, identifying its end vertices and making the graph simple. The resulting graph is denoted as G/e.

The following lemma ensures the existence of contractible edges.[4]

Lemma 1. *Each vertex v of a plane triangulation has an adjacent vertex* a_v *such that* (v, a_v) *is contractible.*

An *expansion* of v is the reverse operation of a contraction, i.e., expanding vertex v of G/e to obtain G.

Note that given straight line embedding of $(G/e)^*$, we can obtain G^* by the expansion of v preserving the positions of the vertices of $(G/e)^*$: Add a new vertex u in the neighborhood of v from where one can see all the vertices adjacent to u in G and redraw the edges incident to u and v obtaining G. We will refer to this special way of expansion of $(G/e)^*$ by 'expansion'.

A plane triangulation G with outer cycle (u, v, w) is *reducible* if it has three disjoint contractible edges $\{(u, a_u), (v, a_v), (w, a_w)\}$ no two of which are sides of a separating tetragon. This tetragon condition is required to ensure that the three edge contractions are compatible in the following procedure: three edges must be simultaneously contracted and simultaneously expanded.

The following algorithm obtains a desired straight line embedding.

procedure EMBEDDING(G);{G is a plane triangulation}
begin
Let u, v, and w be the three vertices on the outer cycle of G;
if G is reducible **then**

begin
Contract three disjoint contractible edges $e = (u, a_u), f = (v, a_v)$, and $g = (w, a_w)$ no two of which are side of a separating tetragon;
$\tilde{G}$ =$G/e/f/g$;
$\tilde{G}^*$:= **EMBEDDING**($\tilde{G}$) ;
Expand the three vertices on the outer triangle of $\tilde{G}^*$ and obtain G^* so that $(a_u, a_v)//(u, v)$ and $(a_v, a_w)//(v, w)$
end;
else{ G is C_3 or $G_{p,q}$ as shown in Fig. 1 (a) or (c), respectively}
Obtain a straight line embedding G^* as shown in Fig. 1. { If G is C_3, the virtual-height and virtual-width of G^* are two. If G is $C_{p,q}$, the virtual-height is $\lfloor \frac{n+2}{2} \rfloor$ and the virtual-width is $\lceil \frac{n+2}{2} \rceil$, where n is the order of G. }
end;

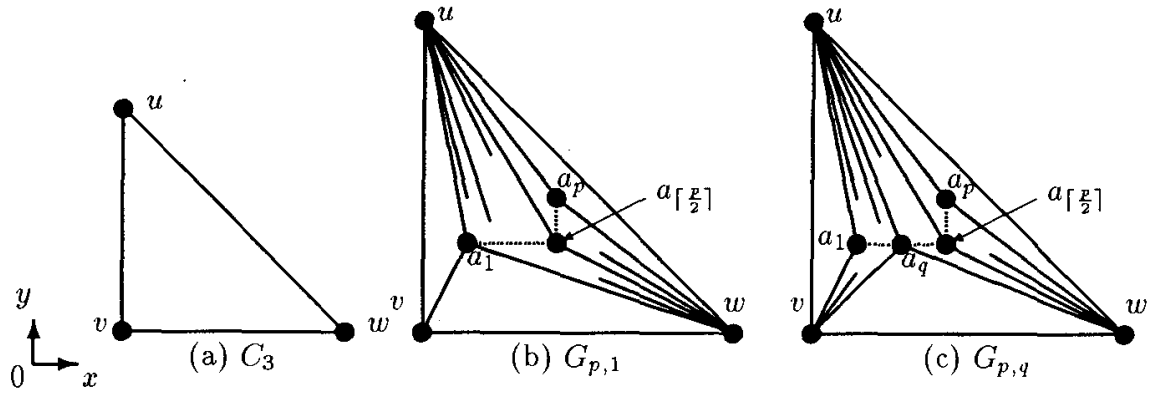


Fig. 1.

3. The Validity of the Algorithm

In this section, we show that **EMBEDDING** always obtains a desired straight line embedding G^* of the given plane triangulation G in polynomial time.

3.1. Irreducible Graphs

First, we show that the tetragon condition can be relaxed.

Lemma 2. *Let G be a plane triangulation of order at least six and let (u, v, w) be the outer triangle of G. G is reducible if and only if there exist three disjoint contractible edges $e = (u, a_u), f = (v, a_v)$, and $g = (w, a_w)$.*

Proof. The 'only if ' part is obvious by the definition of *reducible*.

Assume that G has three disjoint contractible edges $e = (u, a_u), f = (v, a_v)$, and $g = (w, a_w)$. If no two of e, f, and g are sides of a separating tetragon, G is reducible.

Therefore, suppose that there exists a separating tetragon $C_4^1 = (u, a_u, a_v, v)$ in G. Then, we can always find a contractible edge $f' = (v, x)$ such that x is a vertex inside C_4^1 and four vertices $\{u, a_u, x, v\}$ do not form a separating tetragon; v has a neighbor x inside C_4^1, otherwise C_4^1 has a diagonal (u, a_v) and this contradicts the contractibility of e. Let $f' = (v, x)$. If f' is not contractible, we can find a contractible edge (v, x') inside the triangle one side of which is f' by Lemma 1. Let this contractible edge (v, x') be f' newly. If four vertices $\{u, a_u, x', v\}$ form a separating tetragon again, we have to repeat the above procedure for finding new f'. But this procedure will end in finding a desired contractible edge f' since the new separating tetragon always contains less vertices than the old one.

Moreover, if there exists a separating tetragon $C_4^2 = (u, a_u, a_w, w)$, we can find an edge $g' = (w, y)$ inside C_4^2 such that $\{w, y, a_v, v\}$ does not form a separating tetragon.

There is no tetragon in G which contains both edges f' and g (or f' and g') since x is inside C_4^1. Thus, we have obtained a desired three edges $\{e, f', g\}$ (or $\{e, f', g'\}$). □

Now, we will show that G is C_3 or $G_{p,q}$ if it is irreducible. $G_{p,q}$ is a plane triangulation as shown in Fig. 1 (c): All interior vertices are adjacent to u. Let the neighbors of u be $A_u = \{v, a_1, a_2, ..., a_p, w\}$ in order from v to w where $p \geq 1$. For some $q \leq \lceil \frac{p}{2} \rceil$, all members of $\{u, a_1, a_2, ...a_{q-1}, a_q\}$ are adjacent to v and all members of $\{a_q, a_{q+1}, a_{q+2}, ..., a_p, u\}$ are adjacent to v. (Fig. 1 (b) shows $G_{p,1}$, which is a special case of $G_{p,q}$.)

Lemma 3. *Let G be a plane triangulation with outer triangle (u, v, w). If G is irreducible, then G is C_3 or $C_{p,q}$ as shown in Fig. 1 (a) and (c).*

Proof. If the order of G is 3, 4, or 5, then G is C_3,$G_{1,1}$, or $G_{2,1}$, respectively. Then assume that the order of G is 6 or more.

By Lemma 2, we only have to show that G is $G_{p,1}$ or $G_{p,q}$ unless G has three disjoint contractible edges.

Let U_c be the set of neighbors of u such that (u, x) is contractible for $x \in U_c$. V_c and W_c are similarly defined.

By Lemma 1, $| U_c |$, $| V_c |$, and $| W_c |$ are all at least one. Then by Hall's theorem[3] for existence of matchings in bipartite graphs, we need only consider two cases in which G doesn't have three disjoint contractible edges

Case 1) $| U_c \cup V_c | = 1$ (or $| V_c \cup W_c | = 1$ or $| W_c \cup U_c | = 1$):

Let (u, a_u) and (v, a_v) be the unique contractible edges incident to u and v, respectively and $a_u = a_v$. (See Fig. 2(a).) Let the neighbors of u be $A_u = \{v, a_u = a_1, a_2, ..., a_p, w\}$ in order from v to w. (a_u is next to v since (u, a_u, v) forms a triangle and (u, a_u) is contractible.) Since (u, a_2) is not contractible, there exists a

vertex $x \in A_u$ such that (u, a_2, x) is a separating triangle. But, if $x \in A_u - \{v\}$, then by Lemma 1, triangle (u, a_2, x) should have a contractible edge (u, y) inside such that $y \neq a_u$. This contradicts the fact that u has only one contractible edge (u, a_u). Therefore $x = v$. By a similar argument, we can see that all the vertices in $A_u - \{v\} + \{u\}$ are adjacent to v. Since G is a plane triangulation, $G = G_{p,p}$.

Case 2) $| U_c \cup V_c \cup W_c | = 2$:

By symmetry, we can assume that (v, a_v) and (w, a_w) are unique contractible edges incident to v and w, respectively and there are two contractible edges (u, a_v), (u, a_w) incident to u as shown in Fig. 2(b); Otherwise, if each vertex of u, v, and w has only one neighbor, then it is Case 1); and if two vertices, say u and v have two neighbors a_v and a_w in common, then triangle (u, v, a_u) or (u, v, a_w) must be a separating one and it is a contradiction.

Now, let the neighbors of u be $A_u = \{v, a_v = a_1, a_2, ..., a_p = a_w, w\}$ in order from v to w. By a similar argument as in Case 1), we see that all the vertices in $A_u - \{v\} + \{u\}$ are adjacent to v or w. Since G is a plane triangulation, there exists an integer q such that $a_i (i \leq q)$ is adjacent to v and $a_j (j \geq q)$ is adjacent to w.

Without loss of generality, we can assume that $q \leq \lceil \frac{p}{2} \rceil$. Since G is a plane triangulation, $G = G_{p,q}$. □

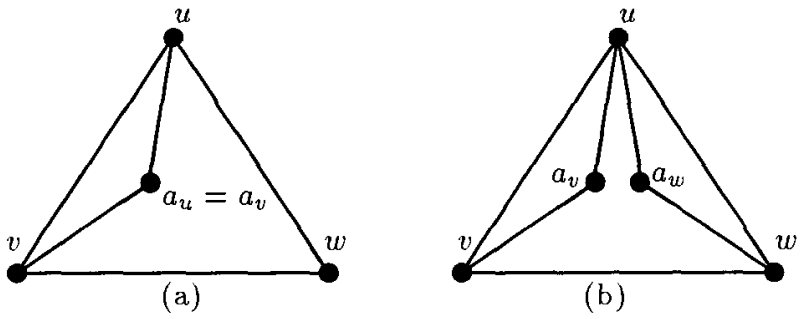


Fig. 2.

3.2. Expansions for Desired Embeddings

We will show that the algorithm always gives a desired embedding.

Lemma 3 says that only C_3 and $G_{p,q}$ $(p \geq 1, q \leq \lceil \frac{p}{2} \rceil)$ are irreducible graphs.

Let m be the number of three edges contractions (i.e. number of iterations) in the procedure. Note that $n - 3m \geq 3$, where n is the order of the original plane triangulation G.

If $n - 3m = 3$, then the expansion phase starts with C_3 to obtain G^*. C_3 has a straight line embedding of virtual-height two and virtual-width two as shown in Fig. 1(a). The procedure expands a straight line embedding with adding three vertices a_u, a_v, and a_w at cost of increasing each virtual dimension by two since $(a_u, a_v)//(u, v)$ and $(a_v, a_w)//(v, w)$. Therefore, for the straight line embedding of

order n given by procedure **EMBEDDING**,

$$[\text{VIRTUAL-HEIGHT}] = 2 + 2m = \frac{2n}{3} \leq \lceil \frac{2n+1}{3} \rceil \tag{1}$$

$$[\text{VIRTUAL-WIDTH }] = 2 + 2m = \frac{2n}{3} \leq \lfloor \frac{2n+1}{3} \rfloor \tag{2}$$

If $n - 3m \geq 4$, then the expansion phase starts with $C_{p,q}$ to obtain G^*. $C_{p,q}$ has a straight line embedding of virtual-height $\lfloor \frac{n+2}{2} \rfloor$ and virtual-width $\lceil \frac{n+2}{2} \rceil$ as shown in Fig. 1 (c). In this case, for the straight line embedding of order n given by procedure **EMBEDDING**,

$$[\text{VIRTUAL-HEIGHT}] = \lfloor \frac{n+2}{2} \rfloor + 2m = \lfloor \frac{n+m+2}{2} \rfloor \leq \lfloor \frac{2n+1}{3} \rfloor \tag{3}$$

$$[\text{VIRTUAL-WIDTH }] = \lceil \frac{n+2}{2} \rceil + 2m = \lceil \frac{n+m+2}{3} \rceil \leq \lceil \frac{2n+1}{3} \rceil \tag{4}$$

We end this subsection illustrating that we can always obtain G^* from $\tilde{G}^*$ so that $(a_u, a_v)//(u, v)$ and $(a_v, a_w)//(v, w)$.

Figure 3 shows how to expand $\tilde{G}^*$ into G^*. In expansion, we add three vertices a_u, a_v, and a_w at proper positions in the neighborhoods of u, v, and w, respectively. In Fig. 3, these neighborhoods are indicated by hatched triangles, say N_u, N_v, and N_w. We can always find desired positions: draw a vertical line intersecting N_u and a horizontal line intersecting N_w in such a way that these two lines intersect in N_v. (These are the dashed lines in Fig. 3.)

Add a_u (a_w) at an arbitrary position in the intersection of vertical (horizontal, respectively) line and N_u (N_w, respectively) and a_v at the intersection of the two lines.

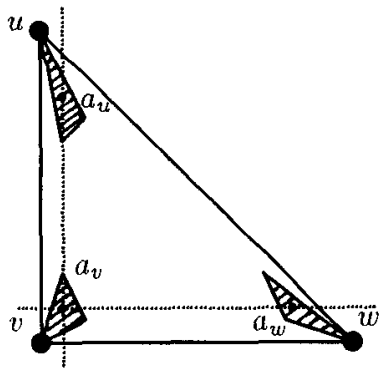


Fig. 3.

3.3. Computational Time Complexity

The overall computational time complexity of the algorithm is essentially determined by the complexity of finding three contractible edges of a plane triangulation

and the complexity of the contractions and expansions, multiplied by the depth of the recursive call. As described in the proof of Lemma 3, we have only to find the neighbors of the vertices of the outer triangle connected by contractible edges for finding three contractible edges. Thus, procedure **EMBEDDING** is totally executed in polynomial time.

4. Concluding Remark

Our discussion started with referring to the grid embedding with the limited grid height and grid width, for which the best known result is $n-2$ by $n-2$ by Schnyder. Then we modified the problem to new one by defining the virtual-height and virtual-width. The virtual-height (virtual-width) is a concept close to the *true* grid height. We conjecture that a straight-line embedding of virtual-height k can be transformed to one of true grid height k at the cost of increasing the true width.

Acknowledgements

This research was supported in part by scientific research Grants-in-Aid from Ministry of Education, Science & Culture, Japan.

References

1. I. Fáry, "On straight line representation of planar graphs", *Acta Sci. Math. (Szeged)* **11**, (1948) 229–233.
2. H. Fraysseix, J. Pach and R. Pollack, "Small sets supporting Fáry embedding of planar graphs", *Proc. 20th ACM Symp. on Theory of Computing* (1988) 426–433.
3. P. Hall, "On representatives of subsets", *Journal of London Mathematics Society* **10**, (1935) 26–30.
4. G. R. Kampen, "Orienting planar graphs", *Discrete Mathematics* **14**, (1976) 337–341.
5. W. Schnyder, "Embedding planar graphs on the grid", *Proc. 1st ACM-SIAM Symp. on Discrete Algorithms* (1990) 138–148.
6. S. K. Stein, "Convex maps", *Proc. Amer. Math. Soc.* **2**, (1951) 464–466.
7. K. Wagner, "Bemerkungen zum vierfarbenproblem", *Jber. Deutsch. Math.-Verein* **46**, (1936) 26–32.

Algorithmic Aspects of VLSI Layout
Eds. M. Sarrafzadeh and D. T. Lee
© World Scientific Publishing Co.

ROUTING AROUND TWO RECTANGLES TO MINIMIZE THE LAYOUT AREA

TEOFILO F. GONZALEZ
Department of Computer Science, University of California, Santa Barbara, CA 93106, USA

and

SINGLING LEE
Department of Computer Science and Information Engineering, National Chung Cheng University Chiayi 62107, Taiwan, ROC

ABSTRACT

The problem of routing n two-terminal nets around two equal-width rectangles to minimize the total area is discussed. We develop an $O(n \log n)$ ($O(n)$ if the set of terminals is initially sorted) time approximation algorithm for this problem. Our algorithm generates a layout with area at most 2 OPT, where OPT is the area of an optimal area layout. We establish a lower bound for the area of an optimal layout from a lower bound for the size of the components, and a lower bound for the area occupied by the wires. The former lower bound is derived from the number of terminals on the sides of the rectangles, and the latter lower bound is based on the number of corners crossed by each wire. This suggests that nets should be connected by wires that "cross" the least number of corners (called *lnc*-wires). Nets for which all their *lnc*-wires cross the same rectangle corners are connected by *lnc*-wires, and a subset of the remaining nets is connected by *lnc*-wires that blend with previously introduced wires. To guarantee our approximation bound the remaining nets are connected in several ways, and a set of layouts is generated. The layout generated by the algorithm is a minimum area layout among these layouts.

Keywords: Rectangle routing, approximation algorithms, Manhattan routing, Knock-Knee routing, efficient algorithms.

1. Introduction

Let T (top) and B (bottom) be two rectangles with equal width ($w_T = w_B$) and possibly different heights (h_T and h_B). Assume the rectangles are placed on the same plane with the same orientation. The left side of T and B is placed along the same vertical line and rectangle T is above rectangle B. The distance between these two rectangles is at least $\lambda > 0$ units and the exact distance will be decided by our routing algorithm. Let N be a set of terminals (or terminal points) that lie on the sides of T and B. Set N is partitioned into n disjoint subsets N_i, $1 \leq i \leq n$,

called *nets*. All the terminals in each net have to be made electrically common by interconnecting them with wires. The wires consist of a finite number of horizontal and vertical segments. All the horizontal segments are assigned to one layer and all the vertical segments are assigned to the other layer. Wire segments on different layers can be connected directly at any given point z by a wire perpendicular to the layers if both wire segments cross point z in their respective layers (i.e., the connection is made through a *contact cut* or *via*). Every pair of (distinct) parallel wire segments must be at least λ units apart and every wire segment must be at least λ units from each side of rectangles T and B, except in the region where the wire connects a terminal in N. Also, no wire segment is allowed inside of T and B on any of the layers. We assume that the distance between any two vertical lines each including a terminal located in the middle channel (the region between the bottom side of rectangle T and the top side of rectangle B) is at least λ. We shall refer to this assumption as the *middle-channel assumption*. Later on we explain why we make this assumption, and show how it can be eliminated at the expense of an additional layer for routing.

Problem $R2M$ (routing around two rectangles) consists placing T and B, vertically aligned, and connecting the terminals in each net by wires, that satisfy the restrictions imposed above, in such a way that the smallest enclosing rectangle, with the same orientation as T and B, has least area amongst all feasible layouts. This problem has applications in the bottom-up layout of integrated circuits.[1,2] The $R2M$ problem is referred to as the $2-R2M$ problem when each net consists of exactly two terminals. In this paper we focus on the $2-R2M$ problem under the wiring model just discussed which is referred to as the *Manhattan wiring model*. We also consider the $2-R2M$ problem without the middle-channel assumption under the *knock-knee* wiring model. Under this wiring model,[3,2] vertical and horizontal segments from different nets may be assigned to the same layer as long as they do not touch, and two wires may bend at a grid point rather than just cross as in the Manhattan wiring model. A Manhattan wiring is also a knock-knee wiring, but the converse is not necessarily true. When we refer to a wiring we mean a wiring under Manhattan model. When we wish to refer to knock-knee wirings we shall refer to them explicitly.

The $2-R2M$ problem without the middle channel assumption is an NP-hard problem because the channel between the two rectangles corresponds to the Manhattan channel routing problem which is known to be NP-hard.[4] It is not known whether the $2-R2M$ is NP-hard. With respect to the knock-knee model the $R2M$ problem without the middle channel assumption is an NP-hard problem because the channel between the two rectangles corresponds to the knock-knee channel routing problem which is known to be NP-hard,[5] but it is not known whether or not the $2-R2M$ with or without the middle-channel assumption or the $R2M$ are NP-hard. As we shall see later on, the reason for introducing the middle-channel assumption is not the complexity of the optimization problem, but rather the complexity of gen-

erating good area layouts for the channel routing problem (routing in the middle channel).

The $R1M$ problem is defined similarly, except that all terminals are located on the sides of one rectangle. Hashimoto and Stevens[6] present an $O(n \log n)$ algorithm to solve the $R1M$ problem for the case when all the points in N lie on one side of a rectangle. An $\Omega(n \log n)$ lower bound of time complexity for this problem has been established.[7] There are several algorithms to solve the $R1M$ problem when all nets have exactly two terminals.[8,9,10,2] Gonzalez and Lee's algorithm[10] is optimal with respect to the time complexity bound. Approximation algorithms for the $R1M$ problem have been developed by Gonzalez and Lee.[11,12] The time complexity for these algorithms is $O(m(n+\log m))$ and the best one[12] generates a layout with area at most 1.6 OPT, where OPT is the area of an optimal layout, m is the number of terminals and n is the number of nets. If more than two layers are allowed and wire overlap is permitted, the $R1M$ problem becomes an NP-hard problem,[13] even when the size of all nets is two.

Chandrasekhar and Breuer[14] studied a restricted version of the $2-R2M$ problem in which some type of nets have to be connected by a special type of wires. This limitation on the set of feasible solutions simplifies the routing problem considerable and makes it polynomially solvable. Unfortunately, an optimal area layout for this problem with the additional restrictions is in general larger than that of an optimal area layout for the unrestricted $2-R2M$ problem. In our problem we allow all possible type of connections and as a result of this we compare our layouts with the area of a "true" optimal area layout. Baker[15] presents an $O(n \log n)$ algorithm for the $2-R2M$ problem in which optimality is measured with respect to the perimeter of the resulting enclosing rectangle. The algorithm generates a layout whose perimeter is within 1.9 times the perimeter of an optimal layout. The perimeter is simpler to approximate than the area mainly because the perimeter is a linear objective function, but layout area is considered to be one of the most important objective functions in VLSI optimization. Our approximation bound is two and Baker's[15] is 1.9; however, Baker[15] does not require the middle-channel assumption. Without the middle-channel assumption we can still approximate within two the layout area, but at the expense of an extra layer in the middle channel. Without the extra layer the problem is difficult to approximate. To convince yourself of this fact let us just consider the middle channel. The best known approximation algorithm for this problem has an approximation bound which is bounded by a constant (the constant is greater than two).[16] Even if such algorithm is used directly, it does not imply that we can generate a constant times optimal wiring because it may use a set of additional columns to the right of the channel, i.e., the algorithm approximates with respect to channel width, but not necessarily with respect to the channel area. This is the main reason we have made the middle-channel assumption. We should note that the middle-channel assumption is not too restrictive, since we show that such assumption may be eliminated by simply adding an extra layer and wiring under the knock-knee model.

Sarrafzadeh and Preparata[2] developed an algorithm that generates optimal area layouts for the case when the area of the layout is defined as the area of two nonoverlapping rectangles that enclose all the wires, i.e., one rectangle encloses T and the other encloses B. Under this objective function the problem can be solved efficiently, but then the resulting building block is not rectangular. This makes the wiring problem in the bottom-up approach difficult to solve. Our building block at each level in the bottom-up approach is a rectangle, which is simpler to handle.

In this paper we present an $O(n \log n)$ ($O(n)$ if the set of terminals is initially sorted) time approximation algorithm for the $2 - R2M$ problem that generates a layout with area at most 2 OPT, where OPT is the area of an optimal layout. In the final section we show that this result also holds when the middle-channel assumption is removed provided knock-knees are allowed and three layers are available for routing.

Hereafter we assume that the terminals points are initially sorted, i.e., the terminal points are given by two lists each corresponding to the order in which terminals appear while traversing each of the rectangles in the clockwise direction starting at the bottom-left corner. We define some terms before outlining our algorithm. A net is called *local* if its terminals are located on the same side of the same rectangle. Otherwise, it is called *global*. If a corner s can be horizontally projected to a wire without intersecting any side of the two rectangles, then the wire *crosses horizontally corner s*. A net is said to be connected by an *lnc*-wire if the wire crosses horizontally the least number of rectangle corners amongst all wires connecting the net. A net is called *simple* if all *lnc*-wires connecting its terminals cross horizontally the same corners of T and B. Otherwise, the net is called *complex*. Note that in the definition for simple nets we use horizontal crossing rather than just crossing. The only place where this makes a difference is in the type of nets shown in Fig. 5. In our algorithm we always wire these nets as in Fig. 5(a).

The first few steps of our procedure correspond to the initial steps of previous algorithms.[9,10] In this step all local nets are connected by *lnc*-wires. The reason why this is a good decision is that any layout can be transformed to another layout without increasing its area in such a way that all local nets are connected by *lnc*-wires. The problem is then reduced to determining the type of wire connecting each global net in the presence of some previously introduced wires. We establish a lower bound for the area of an optimal layout from a lower bound for the size of the components and a lower bound for the area occupied by the wires. The former lower bound is derived from the number of terminals on the sides of the rectangles, and the latter lower bound is based on the number of corners crossed by each wire. This suggests that the global nets should be connected by *lnc*-wires. Simple nets are connected by *lnc*-wires and a subset of the complex nets is connected by *lnc*-wires that blend with previously introduced wires. All remaining unrouted nets have all their terminals on the top and bottom sides of the rectangles. Because the lower bound for the width of the rectangles is the number of terminals divided by four (rather than two for the total height of the rectangles), the lower bound is not large

enough to establish an approximation bound of two when the remaining nets are routed in the obvious way. To achieve this approximation bound, the remaining nets are connected in several ways and a set of layouts is generated. Our algorithm selects the best of these layouts as its output. To establish our approximation bound we find the cost (in terms of area) of transforming an optimal solution to the solution generated by the algorithm. This is done for each type of net separately.

Let $2_S - R2M$ ($2_C - R2M$) denote a $2 - R2M$ problem with the property that each global net is simple (complex). In Sec. 2 we define our notation and present some basic results. In order to simplify the exposition of our results, we begin by presenting approximation algorithms for restricted versions of the $2-R2M$ problem. In Sec. 3 we present an approximation algorithm for the $2_S - R2M$ problem. An approximation algorithm $2_C - R2M$ problem is presented in Sec. 4 and in Sec. 5 we combine these results to obtain our approximation algorithm for the $2 - R2M$ problem.

2. Notation and Basic Results

We begin by defining our notation and proving some lemmas that will simplify our notation. Then we reduce our problem to only having to specify the type of path for the wire connecting each global net. This simplifies our analysis.

The four corners of T and B are labeled as follows. Starting with the bottom-left corner of T (B), traverse the sides of rectangle T (B) clockwise. The ith corner visited is labeled S_{i-1} (R_{i-1}). We use TL, TT, TR, and TB (BL, BT, BR, and BB); T_l, T_t, T_r and T_b (B_l, B_t, B_r and B_b); to represent the left, top, right and bottom sides of $T(B)$, respectively. Let X_jY_k represent the set of nets with one terminal located on side j of rectangle X and the other located on side k of rectangle Y, where $X,Y \in \{T,B\}$ and $j,k \in \{l,r,t,b\}$ (see Fig. A.1 in Appendix A). We use x_jy_k to represent the number of nets in set X_jY_k. For $1 \leq j \leq n$, the pairing function $C(j)$ is defined in such a way that T_j and $T_{C(j)}$ belong to the same net. We establish the following lemma to simplify our remaining notation.

Lemma 1. *Let W be any layout for an instance of the $2 - R2M$ problem. For each net $N_i \in T_bB_x$ (T_xB_t), where $x \in \{t,l,r,b\}$, if net N_i is connected by a wire that crosses the top (bottom) side of rectangle $T(B)$, then there exists another layout M such that net N_i is connected by a wire that does not cross the top (bottom) side of rectangle $T(B)$, and the area of M is not larger than the area of W.*

Proof. The proof is based on a simple interchange argument. Let M be minimum area layout in which the wire for each net crosses the same corners of the rectangles as the one in layout W, except for the wire for the net in Fig. 1(a) which is replaced by the one in Fig. 1(b). It is simple to see that the area of layout M is less than or equal to the area of layout W. □

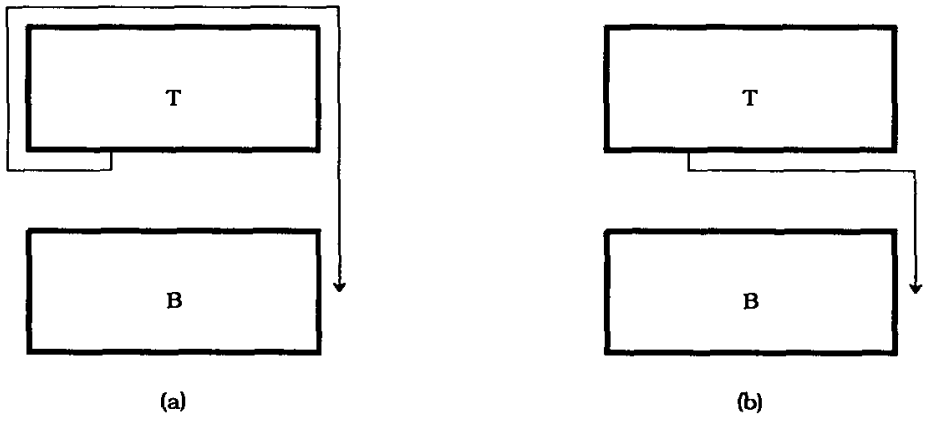


Fig. 1. (a) Layout W, and (b) layout M.

Hereafter, whenever we refer to an optimal area layout we assume, without loss of generality, that it cannot be transformed by applying the interchange argument given in Lemma 1, nor that it has wires with excessive segments (wires which can be trivially replaced by other wires without increasing the layout area). Note that this assumption does not make the problem easier; however, it allows us to use a simpler notation. The layouts constructed by our algorithm also satisfy these properties.

Since we are only concerned with layouts that cannot be transformed by the rule given by Lemma 1 and which do not have excessive segments, the type of wire connecting net $N_j = \{T_i, T_{C(i)}\}$ in a layout can be characterized by a triple (i, x, y), for $x, y \in \{+, -\}$ as follows.

1. If T_i and $T_{C(i)}$ are located on the same rectangle X, then $x \neq y$ and the wire connecting net N_j consists of the following sequence of wire segments: the first wire segment is incident to T_i and it is perpendicular to the side where terminal T_i is located; if $x = \text{`}+\text{'}$ ($x = \text{`}-\text{'}$) this segment is followed by a sequence of wire segments parallel to the boundary segments of X encountered while traversing the boundary of rectangle X in the clockwise (counter-clockwise) direction starting at T_i and ending at $T_{C(i)}$; the final segment is perpendicular to the side where $T_{C(i)}$ is located and it is incident at $T_{C(i)}$.
2. If T_i and $T_{C(i)}$ are located on different rectangles, then the wire is more complex. Let h be a horizontal line that partitions the plane so that each rectangle is in a different half space. The wire connecting the net consists of three wire segments: the top segment, the middle segment (possibly empty) and the bottom segment. Assume without loss of generality that T_i ($T_{C(i)}$) is located on rectangle T (B). The top segment consists of the following sequence of wire segments: the first wire segment is incident to T_i and it is perpendicular to the side where terminal T_i is located; if T_i is located on the bottom side of T, then the wire terminates when it reaches line h,

otherwise if $x = '+'$ ($x = '-'$) the first segment is followed by a sequence of wire segments parallel to the boundary segments of T encountered while traversing the boundary of rectangle T in the clockwise (counter-clockwise) direction starting at T_i and ending at the bottom-right (bottom-left) corner of rectangle T; and the final segment extends the last wire segment until it reaches line h. The bottom segment is similar to the top segment. If the top and bottom segment are both located on the left or the right side of the rectangles, then the middle segment is empty because we add the constraint that both the top and the bottom segment must end at the same point on line h. Otherwise, the middle segment is a horizontal wire segment that overlaps with h and joins the two points on line h where the top and bottom wires segments end.

Set $D = \{(d_1, x_1, y_1), (d_2, x_2, y_2), \ldots, (d_n, x_n, y_n)\}$, where $1 \leq d_i \leq 2n$ and $x_i, y_i \in \{+, -\}$ for $1 \leq i \leq n$, is said to be an *assignment* if $|\{d_1, C(d_1), d_2, C(d_2), \ldots, d_n, C(d_n)\}| = 2n$. Any subset of an assignment is said to be a *partial assignment*. Each tuple in an assignment or partial assignment specifies the type of wire connecting a net. For any $(i, x_j, y_j) \in D$, we say that the type of wire connecting T_i and $T_{C(i)}$ specified by D *crosses terminal* z if the terminal z can be horizontally or vertically projected to it without intersecting any side of the rectangles. A wire *crosses corner* s if s can be horizontally and vertically projected to it without intersecting any side of the rectangles. If corner s can be vertically (horizontally) projected to a wire without intersecting any side of the two rectangles, then this wire *crosses vertically (horizontally) corner* s.

For any assignment (or partial assignment) D we define the *height function* H_D as follows:

$$H_D(X) = \textbf{max}\{\text{number of wires given by } D \text{ that cross horizontally point } z \mid z \text{ is a terminal or a corner located on side } X\}, \text{ for } X \in \{TL, TR, BL, BR\};$$

$$H_D(X) = \textbf{max}\{\text{number of wires given by } D \text{ that cross vertically point } z \mid z \text{ is a terminal or a corner located on side } X\}, \text{ for } X \in \{TT, BB\},$$

$$H_D(TB) = \textbf{max} \text{ number of wires given by } D \text{ that cross vertically point } z \mid z \text{ is a terminal or a corner located on side } TB \cup BT\}, \text{ and}$$

$$H_D(TB) = H_D(BT)\ .$$

We shall refer to $H_D(X)$ as the *height of assignment* D *on side* X. For an assignment D, we define $H_D(LL)$, $H_D(RR)$, and $H_D(TMB)$ as follows:

$$H_D(LL) = \max\{H_D(TL), H_D(BL)\} ,$$
$$H_D(RR) = \max\{H_D(TR), H_D(BR)\} , \text{ and}$$
$$H_D(TMB) = H_D(TT) + H_D(BT) + H_D(BB) .$$

For any assignment D and corner $U \in \{S_0, S_1, S_2, S_3, R_0, R_1, R_2, R_3\}$, we define the functions C_D and CH_D as follows:

$C_D(U) = \max\{$number of wires given by D that a cross corner in set $U\}$, and

$CH_D(U) = \max\{$number of wires given by that cross horizontally a corner in set $U\}$.

It is simple to prove the following relationships between these values:

$$H_D(LL) + H_D(RR) \geq \frac{1}{4} \sum_{\substack{U \text{ is a corner} \\ \text{of } T \text{ or } B}} CH_D(U) , \text{ and}$$

$$H_D(TMB) \geq \frac{1}{2} \sum_{\substack{U \text{ is a corner} \\ \text{of } T \text{ or } B}} C_D(U) .$$

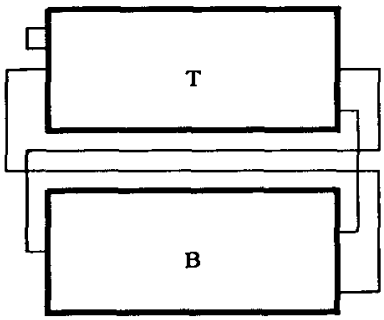
For any assignment D we define

$$h_D = h_T + h_B + \lambda H_D(TMB) , \text{ and}$$
$$w_D = w_T + \lambda(H_D(LL) + H_D(RR)) .$$

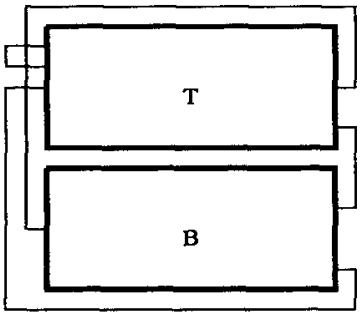
Assignment D is said to be an optimal assignment for problem instance I if D is an assignment with minimum $h_D \cdot w_D$ amongst all assignments for I. Every optimal area layout can be characterized by an optimal assignment, but not every optimal assignment characterizes an optimal area layout. Figure 2 gives two optimal assignments, but only one of them has an optimal area layout. Therefore, not every optimal assignment D can be wired inside a rectangle with area $h_D \cdot w_D$. However, every assignment D can be wired inside a rectangle with area $h_D \cdot (w_D + \lambda)$. Furthermore, such layout can be constructed in $O(n)$ time (remember that the we assumed that set of terminals is sorted). This facts are established in the following two lemmas.

Lemma 2. *For every assignment D, there is a rectangle Q of size at most h by $(w + \lambda)$, where $h = h_T + h_B + \lambda H_D(TMB)$, and $w = w_T + \lambda(H_D(LL) + H_D(RR))$, with the property that rectangle T and B together with the interconnecting wires given by assignment D can be made to fit inside Q.*

Proof. The proof is a direct generalization of the proof for the $R1M$ problem,[9] and the T-shape routing problem.[17] Pinter's procedure[17] may require one additional



(a) Optimal assignment without an optimal area layout.



(b) Optimal assignment with an optimal area layout.

Fig. 2. Optimal assignments and optimal area layouts.

vertical track. For the case (see Fig. 2(a)) one additional vertical track is required on either the left or right side of the rectangles. □

Lemma 3. *For any assignment D a layout with the area given by Lemma 2 can be obtained in $O(n)$ time.*

Proof. The proof of this lemma is a straight forward generalization of the proof for the $R1M$ problem.[9] The algorithm that constructs the final layout uses as a subalgorithm the known algorithms.[7,6,17] □

For an assignment D we define the *area function* $A(D)$ as

$$(h_T + h_B + \lambda H_D(TMB)) \cdot (w_T + \lambda(H_D(LL) + H_D(RR))) .$$

Hereafter, we assume that $A(D)$ is the total area required for a layout of T and B and all interconnections given by D. As we showed before this is not always true, but since the difference is insignificant, we ignore it. Hereafter we corrupt our notation and say that every optimal assignment has an optimal area layout.

Net N_i is said to be a *local net* if its two terminals are located on the same side of the same rectangle, or both are located in the middle channel (see all nets labeled L in Fig. A.1 in Appendix A). Otherwise, net N_i is said to be *global*. Therefore, the set of nets L consists of the following set of nets.

$$L = T_bB_t \cup T_tT_t \cup B_tB_t \cup T_rT_r \cup B_rB_r \cup T_bT_b \cup B_bB_b \cup T_lT_l \cup B_lB_l .$$

For the set of nets L we define ML as the partial assignment in which each local net is connected by an *Inc*-wire. We show that every assignment can be transformed to another assignment without increasing its area and in which all local nets are connected by *Inc*-wires.

Lemma 4. *Every assignment D can be transformed to an assignment M such that $ML \subseteq M$ and $A(M) \leq A(D)$.*

Proof. The proof follows the same lines as the one for the $R1M$ problem.[9,10] □

The $2 - R2M$ problem has been reduced to the problem of specifying the type of wire connecting each global net in the presence of the partial assignment ML, i.e., generate an assignment that includes ML. Remember that once we have an assignment, the proof of Lemma 3 (a constructive proof) can be used to find an optimal area layout for it. In the next two sections we present approximation algorithms for the $2_S - R2M$ problem and $2_C - R2M$ problem. In Sec. 5 we show how to combine these results to obtain our approximation algorithm that generates a layout with area at most $2\,OPT$, where OPT is the area of an optimal area layout.

3. Approximation Algorithm for the $2_S - R2M$ Problem

In this section we present an approximation algorithm for the $2_S - R2M$ problem. First we define the assignment from which our algorithm generates the final layout. In the final layout all simple nets are connected by *lnc*-wires. Then in Lemmas 5–9 we find a bound for the cost of transforming any optimal solution to our solution, and in Lemma 10 we establish a lower bound for the area of an optimal area layout. In Theorem 1 we establish the approximation bound for simple nets based on the previous lemmas.

Let S be the set of global nets for the $2_S - R2M$ problem. By definition all nets in S are simple, i.e., all wires connecting a global net cross exactly the same rectangle corners. It is simple to show that the set S consists of the following subset of nets (see all nets labeled S in Fig. A.1 in Appendix A):

$$S = \left(\bigcup_{\substack{f,g \text{ are two} \\ \text{adjacent sides}}} T_f T_g \cup B_f B_g \right) \cup \left(\bigcup_{\substack{j \in \{l,t,b,r\} \\ k \in \{l,r\}}} T_j B_k \cup T_k B_j \right) .$$

For the set of nets in $L \cup S$, we define assignment MS as follows:

$$MS = ML \cup \{\text{each net in } S \text{ is connected by an } lnc\text{-wire}\} .$$

In Fig. 3 we give two layouts for a net in $T_t T_r$. The one in Fig. 3(a) shows the layout for the assignment constructed by our algorithm. Suppose that this net is routed as in Fig. 3(b) in an optimal assignment D. Let M be D except for the wire in Fig. 3(b) which is of the type given in Fig. 3(a). It is simple to show that:

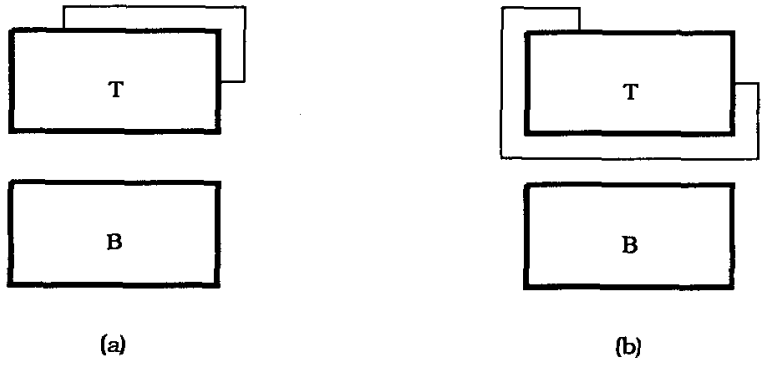


Fig. 3. A wire connecting a net in (a) $T_tT_r - Y_tY_r$, and (b) Y_tY_r.

$$
\begin{aligned}
H_M(TMB) &\leq H_D(TMB)\ , \\
H_M(TL) &= H_D(TL) - 1\ , \\
H_M(TR) &\leq H_D(TR) + 1\ , \\
H_M(BL) &= H_D(BL)\ , \text{ and} \\
H_M(BR) &= H_D(BR)\ .
\end{aligned}
$$

A straight forward generalization of the above observation is given by Lemmas 5–9.

Lemma 5. *Let D be an optimal assignment such that $ML \subseteq D$. Let M be D except that all nets in T_fT_g and B_fB_g, where f and g are two adjacent sides, are assigned as in our algorithm. Then*

$$
\begin{aligned}
H_M(TMB) &\leq H_D(TMB)\ , \\
H_M(TL) &\leq H_D(TL) + \sum_{k\in\{t,b\}} y_k y_l - \sum_{k\in\{t,b\}} y_k y_r\ , \\
H_M(TR) &\leq H_D(TR) - \sum_{k\in\{t,b\}} y_k y_l + \sum_{k\in\{t,b\}} y_k y_r\ , \\
H_M(BL) &\leq H_D(BL) + \sum_{k\in\{t,b\}} z_k z_l - \sum_{k\in\{t,b\}} z_k z_r\ , \text{ and} \\
H_M(BR) &\leq H_D(BR) - \sum_{k\in\{t,b\}} z_k z_l + \sum_{k\in\{t,b\}} z_k z_r\ ,
\end{aligned}
$$

where $Y_fY_g(Z_fZ_g)$ is the set of nets in $T_fT_g(B_fB_g)$ that are connected differently in assignments D and M. (It is important to keep in mind that the set of nets in $T_fT_g - Y_fY_g$ are connected by the same type of wires in D and M.)

Proof. For brevity the proof is not included. The wire connecting a net in set $T_tT_r - Y_tY_r$ and Y_tY_r is illustrated in Fig. 3. □

Lemma 6. *Let D be an optimal assignment such that $ML \subseteq D$. Let M be D except that all nets in T_jB_j, where $j \in \{l, r\}$ are assigned as in our algorithm. Then**

$$
\begin{aligned}
H_M(TMB) &\leq H_D(TMB) \ , \\
H_M(TL) &\leq H_D(TL) + y_l b_l + y_l z_l - y_r b_r - y_r z_r \ , \\
H_M(TR) &\leq H_D(TR) - y_l b_l - y_l z_l + y_r b_r + y_r z_r \ , \\
H_M(BL) &\leq H_D(BL) + t_l z_l + y_l z_l - t_r z_r - y_r z_r \ , \text{ and} \\
H_M(BR) &\leq H_D(BR) - t_l z_l - y_l z_l + t_r z_r + y_r z_r \ ,
\end{aligned}
$$

where T_jZ_j (Y_jB_j) is the set of nets in T_jB_j that are connected differently only on the bottom (top) rectangle in assignments D and M, and Y_jZ_j is the set of nets in T_jB_j that are connected differently on the top and bottom rectangle in assignments D and M.

Proof. Since the proof is straight forward, it is omitted. The wire connecting a net in set (a) $T_lB_l - (T_lZ_l \cup Y_lB_l \cup Y_lZ_l)$, (b) T_lZ_l, (c) Y_lB_l, and (d) Y_lZ_l, are illustrated in Fig. 4. □

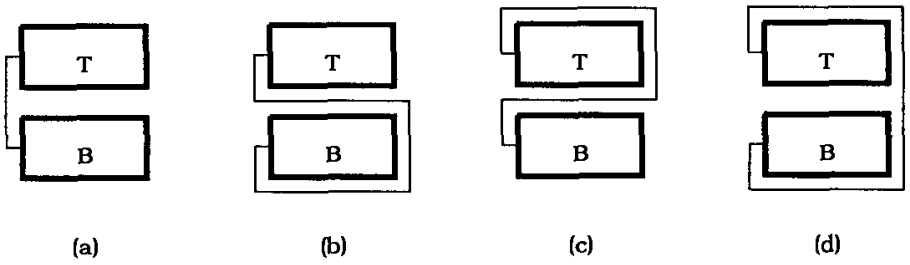


Fig. 4. A wire connecting a net in (a) $T_lB_l - (T_lZ_l \cup Y_lB_l \cup Y_lZ_l)$, (b) T_lZ_l, (c) Y_lB_l, and (d) Y_lZ_l.

Lemma 7. *Let D be an optimal assignment such that $ML \subseteq D$. Let M be D except that all nets in $T_lB_r \cup T_rB_l$ are assigned as in our algorithm. Then*

$$
\begin{aligned}
H_M(TMB) &= H_D(TMB) - 2y_l z_r - 2y_r z_l \ , \\
H_M(TL) &\leq H_D(TL) + y_l b_r + y_l z_r - y_r b_l - y_r z_l \ , \\
H_M(TR) &\leq H_D(TR) - y_l b_r - y_l z_r + y_r b_l + y_r z_l \ , \\
H_M(BL) &\leq H_D(BL) - t_l z_r - y_l z_r + t_r z_l + y_r z_l \ , \text{ and} \\
H_M(BR) &\leq H_D(BR) + t_l z_r + y_l z_r - t_r z_l - y_r z_l \ ,
\end{aligned}
$$

where T_jZ_k (Y_jB_k) is the set of nets in T_jB_k that are connected differently only on the bottom (top) rectangle in assignments D and M, and Y_jZ_k is the set of nets in

*One can trivially establish a sharper bound; however, such bound is not needed to establish the approximation bound of two.

T_jB_k that are connected differently on the top and bottom rectangle in assignments D and M.

Proof. Since the proof is straight forward, it is omitted. The wire connecting a net in set (a) $T_lT_r - (T_lZ_r \cup Y_lB_r \cup Y_lZ_r)$, (b) T_lZ_r, (c) Y_lB_r, and (d) Y_lZ_r, is illustrated in Fig. 5. □

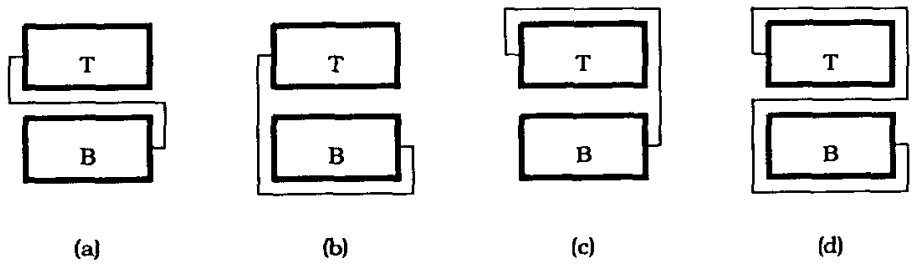


Fig. 5. A wire connecting a net in (a) $T_lB_r - (T_lZ_r \cup Y_lB_r \cup Y_lZ_r)$, (b) T_lZ_r, (c) Y_lB_r, and (d) Y_lZ_r.

Lemma 8. *Let D be an optimal assignment such that $ML \subseteq D$. Let M be D except that all nets in $T_bB_j \cup T_jB_t$, where $j \in \{l,r\}$, which are assigned as in our algorithm. Then*

$$
\begin{aligned}
H_M(TMB) &\le H_D(TMB)\ , \\
H_M(TL) &\le H_D(TL) + y_lz_t - y_rz_t\ , \\
H_M(TR) &\le H_D(TR) - y_lz_t + y_rz_t\ , \\
H_M(BL) &\le H_D(BL) + y_bz_l - y_bz_r\ , \text{ and} \\
H_M(BR) &\le H_D(BR) - y_bz_l + y_bz_r\ ,
\end{aligned}
$$

where Y_jZ_k is the set of nets in T_jB_k that are connected differently in assignments D and M.

Proof. The proof of this lemma is similar to Lemma 6. For brevity the proof is not included. The wire connecting a net in set (a) $T_bT_l - (Y_bZ_l)$, and (b) Y_bZ_l, is illustrated in Fig. 6. □

Lemma 9. *Let D be an optimal assignment such that $ML \subseteq D$. Let M be D except that all nets in $T_tB_j \cup T_jB_b$, where $j \in \{l,r\}$, which are assigned as in our algorithm. Then*

$$
\begin{aligned}
H_M(TMB) &\le H_D(TMB)\ , \\
H_M(TL) &\le H_D(TL) + y_tb_l + y_tz_l - y_tb_r - y_tz_r + y_lb_b + y_lz_b - y_rb_b - y_rz_b\ , \\
H_M(TR) &\le H_D(TR) - y_tb_l - y_tz_l + y_tb_r + y_tz_r - y_lb_b - y_lz_b + y_rb_b + y_rz_b\ , \\
H_M(BL) &\le H_D(BL) + t_tz_l + y_tz_l - t_tz_r - y_tz_r + t_lz_b + y_lz_b - t_rz_b - y_rz_b\ , \text{ and} \\
H_M(BR) &\le H_D(BR) + t_tz_l - y_tz_l + t_tz_r + y_tz_r - t_lz_b - y_lz_b + t_rz_b + y_rz_b\ ,
\end{aligned}
$$

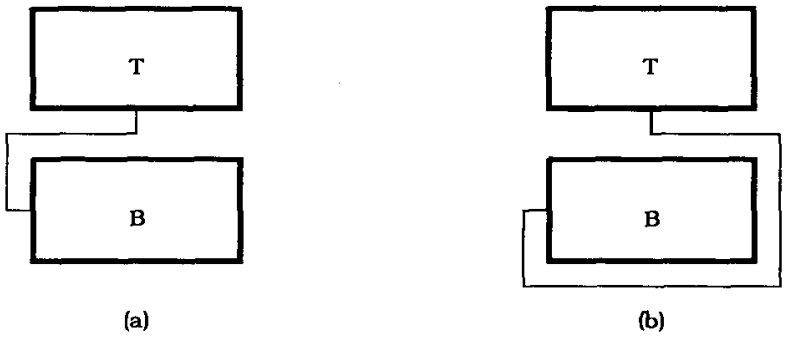


Fig. 6. A wire connecting a net in (a) $T_bB_l - (Y_bZ_l)$, and (b) Y_bZ_l.

where T_jZ_k (Y_jB_k) is the set of nets in T_jB_k that are connected differently only on the bottom (top) rectangle in assignments D and M, and Y_jZ_k is the set of nets in T_jB_k that are connected differently on the top and bottom rectangle in assignments D and M.

Proof. The proof of this lemma is similar to Lemma 6. For brevity the proof is not included. The wire connecting a net in set (a) $T_tT_l - (T_tZ_l \cup Y_tB_l \cup Y_tZ_l)$, (b) T_tZ_l, (c) Y_tB_l, and (d) Y_tZ_l, as illustrated in Fig. 7. □

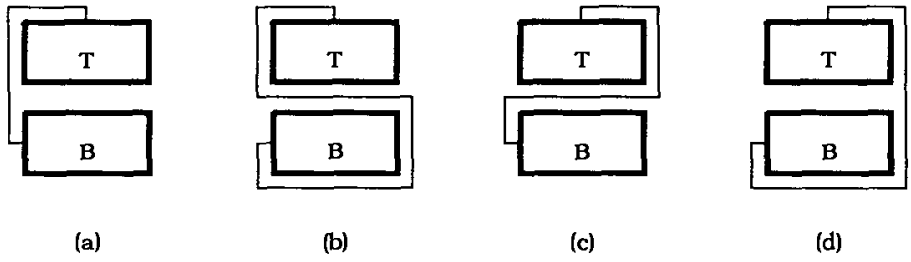


Fig. 7. A wire connecting a net in (a) $T_tB_l - (T_tZ_l \cup Y_tB_l \cup Y_tZ_l)$, (b) T_tZ_l, (c) Y_tB_l, and (d) Y_tZ_l.

Before proving our main result in this section we establish a lower bound for the area of an optimal area layout for the nets in set S. Note that from Lemmas 5–9 we know that our algorithm generates an optimal layout with respect to the height of the enclosing rectangle for the $2_S - R2M$ problem. The lower bounds are given in Table 1.

Lemma 10. *Let D be an optimal assignment such that $ML \subseteq D$. Assignment D, and rectangles T and B satisfy the lower bound given in Table 1.*

Proof. We only prove the lower bounds in the first column of Table 1, since the proof for the bounds in the second column is similar. The proof for these bounds can be obtained by adding the lower bounds given in Table 2. Therefore, we only

Table 1. Lower bounds for the $2_S - R2M$ problem.

set	Contribution to our lower bound for		
	$\frac{w_T}{\lambda} + H_D(LL) + H_D(RR)$	$\frac{h_T+h_B}{\lambda} + H_D(TMB)$	
T_fT_g	$\frac{1}{2}(t_ft_g + y_fy_g)$	$t_ft_g + y_fy_g$	f,g are adjacent
B_fB_g	$\frac{1}{2}(b_fb_g + z_fz_g)$	$b_fb_g + z_fz_g$	f,g are adjacent
T_jB_j	$\frac{1}{2}(t_jb_j + t_jz_j + y_jb_j) + y_jz_j$	$t_jb_j + 2(t_jz_j + y_jb_j + y_jz_j)$	$j \in \{l,r\}$
T_lB_r	$\frac{1}{2}(t_lb_r + t_lz_r + y_lb_r) + y_lz_r$	$2(t_lb_r + y_lz_r)$	—
T_rB_l	$\frac{1}{2}(t_rb_l + t_rz_l + y_rb_l) + y_rz_l$	$2(t_rb_l + y_rz_l)$	—
T_bB_j	$\frac{1}{2}(t_bb_j + y_bz_j)$	$t_bb_j + y_bz_j$	$j \in \{l,r\}$
T_jB_t	$\frac{1}{2}(t_jb_t + y_jz_t)$	$t_jb_t + y_jz_t$	$j \in \{l,r\}$
T_tB_j	t_tb_j	$t_tb_j + t_tz_j + y_tb_j + y_tz_j$	$j \in \{l,r\}$
T_jB_b	t_jb_b	$t_jb_b + t_jz_b + y_jb_b + y_jz_b$	$j \in \{l,r\}$

need to establish the lower bounds given in Table 2. To derive these bounds, we make the following observations.

(a) Each net in T_fT_g (B_fB_g), where f and g are adjacent sides, has exactly one terminal on the top or bottom side of T (B).
(b) Each net in T_jB_k (T_kB_j), where $j \in \{t,b\}$ and $k \in \{l,r\}$, has exactly one terminal on the top or bottom side of T (B).

By the above observations and the fact that every terminal is at least λ units away from each corner of T and B, we know that

$$\frac{w_T}{\lambda} \geq \frac{1}{4}\left(\sum_{\substack{f,g \text{ are} \\ \text{adjacent}}} (t_ft_g + b_fb_g) + \sum_{\substack{j\in\{t,b\}, \\ k\in\{l,r\}}} (t_jb_k + t_kb_j)\right).$$

This lower bound is given by the first column of Table 2.

Let us now establish a lower bound for the number of wires crossing the corners of T and B. For an optimal assignment D, we know that:

1. For adjacent sides f and g, every wire connecting a net in Y_fY_g (Z_fZ_g) crosses horizontally three corners of T (B) and every wire connecting a net in $T_fT_g - Y_fY_g$ ($B_fB_g - Z_fZ_g$) crosses horizontally one corner of T (B).
2. For $j \in \{l,r\}$, every wire connecting a net in $T_jZ_j \cup Y_jB_j$, crosses horizontally four corners of T and B; every wire connecting a net in $Y_lZ_l \cup Y_rZ_r$ crosses horizontally six corners of T and B; and every wire connecting a net in $T_jB_j - (T_jZ_j \cup Y_jB_j \cup Y_jZ_j)$ crosses horizontally two corners of T and B.

Table 2. Lower bounds for the $2_S - R2M$ problem.

set	Contribution to our lower bound for		
	$\frac{w_T}{\lambda}$	$H_D(LL) + H_D(RR)$	
$T_f T_g$	$\frac{1}{4}t_f t_g$	$\frac{1}{4}t_f t_g + \frac{1}{2}y_f y_g$	f, g are adjacent
$B_f B_g$	$\frac{1}{4}b_f b_g$	$\frac{1}{4}b_f b_g + \frac{1}{2}z_f z_g$	f, g are adjacent
$T_j B_j$	—	$\frac{1}{2}(t_j b_j + t_j z_j + y_j b_j) + y_j z_j$	$j \in \{l, r\}$
$T_l B_r$	—	$\frac{1}{2}(t_l b_r + t_l z_r + y_l b_r) + y_l z_r$	—
$T_r B_l$	—	$\frac{1}{2}(t_r b_l + t_r z_l + y_r b_l) + y_r z_l$	—
$T_b B_j$	$\frac{1}{4}t_b b_j$	$\frac{1}{4}t_b b_j + \frac{1}{2}y_b z_j$	$j \in \{l, r\}$
$T_j B_t$	$\frac{1}{4}t_j b_t$	$\frac{1}{4}t_j b_t + \frac{1}{2}y_j z_t$	$j \in \{l, r\}$
$T_t B_j$	$\frac{1}{4}t_t b_j$	$\frac{3}{4}t_t b_j$	$j \in \{l, r\}$
$T_j B_b$	$\frac{1}{4}t_j b_b$	$\frac{3}{4}t_j b_b$	$j \in \{l, r\}$

3. Every wire connecting a net in $T_l Z_r \cup T_r Z_l \cup Y_l B_r \cup Y_r B_l$, crosses horizontally four corners of T and B; every wire connecting a net in $Y_l Z_r \cup Y_r Z_l$ crosses horizontally six corners of T and B; and every wire connecting a net in

$$(T_l B_r - (T_l Z_r \cup Y_l B_r \cup Y_l Z_r)) \cup (T_r B_l - (T_r Z_l \cup Y_r B_l \cup Y_r Z_l))$$

crosses horizontally two corners of T and B.

4. For $j \in \{l, r\}$, every wire connecting a net in $Y_b Z_j \cup Y_j Z_t$ crosses horizontally three corners of T and B and every wire connecting a net in $(T_b B_j - Y_b Z_j) \cup (T_j B_t - Y_j Z_t)$ crosses horizontally one corner of T and B.
5. For $j \in \{l, r\}$, every wire connecting a net in $T_t B_j \cup T_j B_b$, crosses horizontally three corners of T and B.

From the above observations (the ith observation implies the ith line below) we know that:

$$\begin{aligned}
&\sum_{i=0}^{3}(CH_D(S_i) + CH_D(R_i)) \\
&\quad \geq \sum_{\substack{f,g \text{ are} \\ \text{adjacent}}} (t_f t_g + b_f b_g) + \sum_{\substack{f,g \text{ are} \\ \text{adjacent}}} 2(y_f y_g + z_f z_g) \\
&\quad + \sum_{j \in \{l,r\}} 2t_j b_j + \sum_{j \in \{l,r\}} (2(t_j z_j + y_j b_j) + 4y_j z_j) \\
&\quad + 2(t_l b_r + t_l z_r + y_l b_r) + 4y_l z_r + 2(t_r b_l + t_r z_l + y_r b_l) + 4y_r z_l
\end{aligned}$$

$$+ \sum_{j \in \{l,r\}} (t_b b_j + t_j b_t) + \sum_{j \in \{l,r\}} 2(y_b z_j + y_j z_t)$$

$$+ \sum_{j \in \{l,r\}} 3(t_t b_j + t_j b_b)$$

Since $H_D(LL)$ ($H_D(RR)$) is at least as large as 1/4 times the sum of the horizontal height of the four left (right) corners of T and B, we know that

$$H_D(LL) + H_D(RR) \geq \frac{1}{4}(CH_D(R_0) + CH_D(R_1) + CH_D(S_0) + CH_D(S_1))$$

$$+ \frac{1}{4}(CH_D(R_2) + CH_D(R_3) + CH_D(S_2) + CH_D(S_3)) .$$

The lower bounds in the second column of Table 2 follows from the above inequalities. The proof for the lower bounds for the first column of Table 1 is obtained by adding the bounds given by Table 2. This completes the proof for the lemma. □

Theorem 1. *For the $2_S - R2M$ problem, let D be an optimal assignment such that $ML \subseteq D$ and let MS be the assignment generated by our algorithm. Then, $A(MS) \leq 2A(D)$.*

Proof. There is a simpler proof for this theorem, but in order to facilitate its incorporation in Sec. 5, we prove it in four cases.

Case 1: $H_{MS}(LL) = H_{MS}(TL)$ and $H_{MS}(RR) = H_{MS}(BR)$.

From Lemmas 5–10,† we know that $\frac{A(MS)}{A(D)} \leq (1 + \frac{p}{q}) \cdot (1 + \frac{r}{s})$, where

$$p = \sum_{k \in \{t,b\}} y_k y_l + \sum_{k \in \{t,b\}} z_k z_r$$
$$+ y_l b_l + t_r z_r + t_l z_r + y_l b_r + 2y_l z_r + y_b z_r + y_l z_t + y_t b_l + t_t z_r + y_l b_b + t_r z_b$$

$$q = \sum_{\substack{f,g \text{ are} \\ \text{adjacent}}} \frac{1}{2}(t_f t_g + y_f y_g) + \sum_{\substack{f,g \text{ are} \\ \text{adjacent}}} \frac{1}{2}(b_f b_g + z_f z_g)$$

$$+ \sum_{j \in \{l,r\}} (\frac{1}{2}(t_j b_j + t_j z_j + y_j b_j) + y_j z_j) + \frac{1}{2}(t_l b_r + t_l z_r + y_l b_r) + y_l z_r$$

$$+ \frac{1}{2}(t_r b_l + t_r z_l + y_r b_l) + y_r z_l + \sum_{j \in \{l,r\}} \frac{1}{2} \cdot (t_b b_j + y_b z_j)$$

$$+ \sum_{j \in \{l,r\}} \frac{1}{2} \cdot (t_j b_t + y_j z_t) + \sum_{j \in \{l,r\}} t_t b_j + \sum_{j \in \{l,r\}} t_j b_b$$

† p and r are from Lemmas 5–9, and q and s are from Lemma 10.

$$r = -2y_l z_r - 2y_r z_l$$

$$s = \sum_{\substack{f,g \text{ are} \\ \text{adjacent}}} (t_f t_g + y_f y_g) + \sum_{\substack{f,g \text{ are} \\ \text{adjacent}}} (b_f b_g + z_f z_g)$$

$$+ \sum_{j \in \{l,r\}} (t_j b_j + 2(t_j z_j + y_j b_j + y_j z_j)) + 2(t_l b_r + y_l z_r) + 2(t_r b_l + y_r z_l)$$

$$+ \sum_{j \in \{l,r\}} (t_b b_j + y_b z_j) + \sum_{j \in \{l,r\}} (t_j b_t + y_j z_t)$$

$$+ \sum_{j \in \{l,r\}} (t_t b_j + t_t z_j + y_t b_j + y_t z_j) + \sum_{j \in \{l,r\}} (t_j b_b + t_j z_b + y_j b_b + y_j z_b)$$

A straight forward manipulation of the above inequalities[11,12] gives the bound of two.

Case 2: $H_{MS}(LL) = H_{MS}(BL)$ and $H_{MS}(RR) = H_{MS}(TR)$.

The proof for this case is similar to the proof for case 1.

Case 3: $H_{MS}(LL) = H_{MS}(TL)$ and $H_{MS}(RR) = H_{MS}(TR)$.

The proof for this case is trivial since assignment D is optimal, i.e.,

$$H_{MS}(TMB) \leq H_D(TMB) \text{ , and } H_{MS}(TL) + H_{MS}(TR) \leq H_D(TL) + H_D(TR) \text{ .}$$

Case 4: $H_{MS}(LL) = H_{MS}(BL)$ and $H_{MS}(RR) = H_{MS}(BR)$.

The proof for this case is similar to the proof for case 3. This completes the proof of the lemma. □

4. Approximation Algorithm for the $2_C - R2M$ Problem

In this section we present an approximation algorithm for the $2_C - R2M$ problem. Let C be the set of global nets. By definition all the nets in C are complex, i.e., each net can be connected by at least two *Inc*-wires that cross different rectangle corners (see nets labeled C in Fig. A.1 in Appendix A). Clearly,

$$C = \left(\bigcup_{\substack{f,g \text{ are two} \\ \text{opposite sides}}} T_f T_g \cup B_f B_g \right) \cup \left(\bigcup_{k \in \{t,b\}} T_t B_k \cup T_k B_b \right) .$$

In subsection 4.1 we outline our procedure to route the set of nets in $V = T_l T_r \cup B_l B_r$ and establish our approximation bound for that case. We show in

subsection 4.2 how to route the remaining nets, and establish our approximation bound for the $2_C - R2M$ problem.

4.1. Assignment for the Set of Nets $V = T_lT_r \cup B_lB_r$ and the Analysis of its Approximation Bound

We only explain how the assignment $MV(T_lT_r)$ for all nets in T_lT_r is constructed, since the assignment $MV(B_lB_r)$ is constructed by a similar procedure. If the number of nets in T_lT_r is odd then delete one of the nets to make the cardinality of the set even. When the layout for all nets (except for the ones deleted which is at most two) has been constructed, the remaining nets is connected in all possible ways (this number is bounded by a constant) and the best layout is the one generated by the algorithm. The approximation bound will still hold because the nets deleted at this step are connected as in the optimal area assignment in one of the layouts. Figure 8a and 8b give a layout for the assignment $MV(T_lT_r)$ and $MV(B_lB_r)$ constructed by our algorithm for sets T_lT_r and B_lB_r each with four nets. For any permutation, π, of the nets in set T_lT_r we define an assignment ASG(T_lT_r, π) as follows: the wire connecting the kth net ($1 \leq k \leq t_lt_r$) in π, for k is even (odd), begins on the right (left) side of T, it crosses the bottom (top) side of T and ends on the left (right) side of T. Note that by the bottom side of T we mean the middle channel. We claim that there is a permutation, π, of the nets in set T_lT_r such that there is a layout for assignment ASG(T_lT_r, π) with the property that for each k ($1 < k \leq t_lt_r$) the wires connecting the kth and $(k-1)$st net in π can share the same track on the side where the wire connecting the $(k-1)$st net ends. In this case we say that π is a *good permutation* for the set of nets in T_lT_r.

Claim: There is a good permutation for the set of nets T_lT_r.

Proof. For brevity the proof is omitted. An interested reader can find the proof Gonzalez and Lee's paper.[12] □

A good permutation, π, can be constructed by a simple recursive procedure.[12] Once π is obtained, the assignment $MV(T_lT_r)$ can be easily constructed. Figure 8a and 8b give a layout for the assignment $MV(T_lT_r)$ and $MV(B_lB_r)$ constructed by our procedure for sets with four nets each.

Our approximation algorithm constructs the assignment $MV = ML \cup MV(T_lT_r) \cup MV(B_lB_r)$. Before proving that our algorithm generates a layout with area at most twice the area of an optimal layout, we need to establish upper bounds for the area of the layouts obtained from assignment MV and prove lower bounds for the area of an optimal area layout.

Lemma 11. *Let D be an optimal assignment such that $ML \subseteq D$. Let M be D except that all nets in $V = T_lT_r \cup B_lB_r$, which are assigned as in our algorithm. Then*

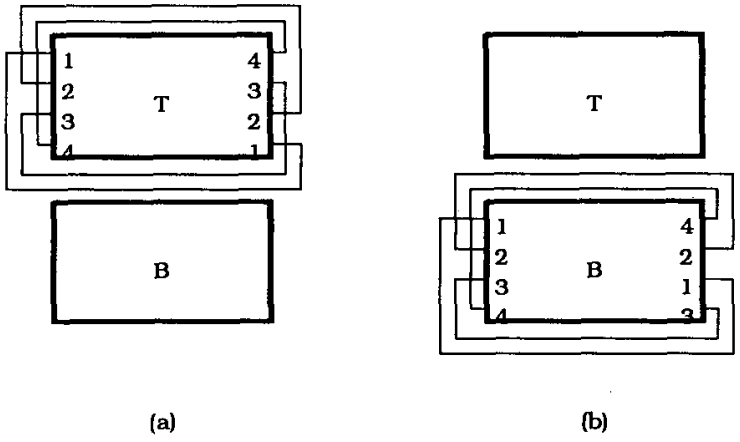


Fig. 8. (a) Layout for set T_lT_r, and (b) layout for set B_lB_r.

$$
\begin{aligned}
H_M(TMB) &= H_D(TMB)\ , \\
H_M(TL) &\le H_D(TL) + \frac{1}{2}t_l t_r + 1\ , \\
H_M(TR) &\le H_D(TR) + \frac{1}{2}t_l t_r\ , \\
H_M(BL) &\le H_D(BL) + \frac{1}{2}b_l b_r + 1\ , \text{ and} \\
H_M(BR) &\le H_D(BR) + \frac{1}{2}b_l b_r\ .
\end{aligned}
$$

Proof. For brevity the proof is not included. □

Before proving our main result in this subsection we establish a lower bound on the area required by an optimal layout. The lower bound is given in Table 3.

Table 3. Lower bounds for the set of nets $T_lT_r \cup B_lB_r$.

set	Contribution to our lower bound for	
	$\frac{w_T}{\lambda} + H_D(LL) + H_D(RR)$	$\frac{h_T+h_B}{\lambda} + H_D(TMB)$
T_lT_r	$\frac{1}{2}t_l t_r$	$2t_l t_r$
B_lB_r	$\frac{1}{2}b_l b_r$	$2b_l b_r$
—	1	2

Lemma 12. *Let D be an optimal assignment such that $ML \subseteq D$. Assignment D and rectangles T and B satisfy the lower bound given in Table 3.*

Proof. Since the proof for the bounds is similar to Lemma 10, it is omitted. We should note that the constants are introduced because every terminal must be located at least λ units from each corner of the rectangles, horizontally there is only one rectangle, and vertically there are two rectangles. □

Theorem 2. *For the $2_C - R2M$ problem, let D be an optimal assignment such that $ML \subseteq D$ and let MV be the assignment generated by our algorithm. Then $A(MV) \leq 2A(D)$.*

Proof. There are four cases that need to be considered:

Case 1: $H_{MV}(LL) = H_{MV}(TL)$ and $H_{MV}(RR) = H_{MV}(BR)$.

From Lemmas 11 (p and q) and 12 (r and s), we know that

$$\frac{A(MV)}{A(D)} \leq \left(1 + \frac{p}{q}\right) \cdot \left(1 + \frac{r}{s}\right) ,$$

where $p = \frac{1}{2}(t_l t_r + b_l b_r) + 1$; $q = \frac{1}{2}(t_l t_r + b_l b_r) + 1$; r=0; and $s = 2(t_l t_r + b_l b_r) + 2$. Since the remaining part of the proof for case 1 is simple, it is omitted.

Case 2: $H_{MV}(LL) = H_{MV}(BL)$ and $H_{MV}(RR) = H_{MV}(TR)$.

The proof for this case is symmetric to the one for case 1.

Case 3: $H_{MV}(LL) = H_{MV}(TL)$ and $H_{MV}(RR) = H_{MV}(TR)$.

From Lemmas 11 (p and q) and 12 (r and s), we know that

$$\frac{A(MV)}{A(D)} \leq \left(1 + \frac{p}{q}\right) \cdot \left(1 + \frac{r}{s}\right) ,$$

where $p = t_l t_r + 1$; $q = t_l t_r + 1$; $r = 0$; and $s = 2(t_l t_r + b_l b_r) + 2$. Since the remaining part of the proof for case 3 is simple, it is be omitted.

Case 4: $H_{MV}(LL) = H_{MV}(BL)$ and $H_{MV}(RR) = H_{MV}(BR)$.

The proof for this case is symmetric to the one for case 3. This completes the proof of the theorem. □

4.2. Assignment for the Remaining Nets, $H = T_t T_b \cup B_t B_b \cup T_t B_b \cup T_t B_t \cup T_b B_b$, and Our Analysis for the Assignment Constructed for the $2_C R2M$ Problem

Remember that L is the set of local nets and that the set of nets C is partitioned into sets $V = T_l T_r \cup B_l B_r$ and $H = T_t T_b \cup B_t B_b \cup T_t B_b \cup T_t B_t \cup T_b B_b$. Remember

that ML is the assignment constructed by our algorithm for the set of nets in L, MV represents the assignment constructed by our algorithm for the set of nets $L \cup V$, and $ML \subseteq MV$.

All remaining unrouted nets have all their terminals on the top and bottom sides of the rectangles. Because the lower bound for the width of the rectangles is the number of terminals divided by four (rather than two for the total height of the rectangles), the lower bound is not large enough to establish an approximation bound of two when the remaining nets are routed as in the previous subsection. To achieve the approximation bound of two, the remaining nets are connected in several ways and a set of layouts is generated. Our algorithm selects the best of these layouts as its output.

Let us now construct the assignment MH for $L \cup V \cup H$ that includes the partial assignment MV. Our algorithm constructs a set of assignments and then selects one with least area, i.e., least $A(\cdot)$, as MH. Let a (b) be the number of nets in H with a terminal located on the top (bottom) side of rectangle T (B). We only deal with the case when

$$H_{MV}(TL) + H_{MV}(TR) + a \geq H_{MV}(BL) + H_{MV}(BR) + b \ ,$$

since the other case can be treated similarly. We construct assignment I, for $0 \leq I \leq a$, as follows. Let $tl = I$ and $tr = a - tl$. The nets in H with a terminal located on the top side of rectangle T are routed as follows on the top rectangle. The leftmost tl nets (those nets in H whose terminal located on the top side of rectangle T is among the (tl)th closest to the top left corner of T when considering only nets in H) are connected by wires that cross the left side of T and the remaining tr nets are connected by wires that cross the right side of T. The nets with a terminal located on the bottom side of rectangle B, are routed on the bottom rectangle by following a similar procedure. In this case we use bl and br, which are determined by procedure FIND defined below. The idea behind the procedure is to find the values for bl and br such that the assignment has minimum width and, as a secondary objective bl and br have values as close to each other as possible. Note that a net could be routed through the left side of rectangle T and through the right side of rectangle B. In this case one needs to add a horizontal wire on the middle channel to join the previously introduced wires. For simplicity of presentation, in what follows we assume that the value of b is even. When the value of b is odd, we need to construct two assignments (since there are two assignments in which bl and br differ by one) whenever we have to deal with balanced cases (e.g., Fig. 9(a)), and then we select the best of these two assignments. Let us now define procedure FIND which determines the values for bl and br from tl and tr.

PROCEDURE FIND (tl,tr,bl,br);
/* Given tl and tr, compute the values for bl and br. */
/* Remember that we are assuming that

$$H_{MV}(TL) + H_{MV}(TR) + a \geq H_{MV}(BL) + H_{MV}(BR) + b \ ,$$

and that the value of b is even. As pointed out earlier, the other cases can be treated similarly. */
$a_1 \leftarrow tl$; $a_2 \leftarrow tr$;
case 1: $H_{MV}(LL) = H_{MV}(TL)$ and $H_{MV}(RR) = H_{MV}(TR)$.
$b_2 \leftarrow 2\min\{a_1 + H_{MV}(TL) - H_{MV}(BL), H_{MV}(TR) + a_2 - H_{MV}(BR)\}$;
if $b \leq b_2$ **then** $bl \leftarrow br \leftarrow b/2$ /* Fig. 9(a) */
else $b_1 \leftarrow b - b_2$
if $a_1 + H_{MV}(TL) = b_2/2 + H_{MV}(BL)$
then $bl \leftarrow b_2/2$; $br \leftarrow b_1 + b_2/2$ /* Fig.9(b) */
else $bl \leftarrow b_1 + b_2/2$; $br \leftarrow b_2/2$ /* Fig. 9(c) */

case 2: $H_{MV}(LL) = H_{MV}(BL)$ and $H_{MV}(RR) = H_{MV}(BR)$.
/* The code for cases 2-4 is omitted since it is similar to the one for case 1. */

case 3: $H_{MV}(LL) = H_{MV}(BL)$ and $H_{MV}(RR) = H_{MV}(TR)$.

case 4: $H_{MV}(LL) = H_{MV}(TL)$ and $H_{MV}(RR) = H_{MV}(BR)$.

END-OF-PROCEDURE FIND

Let D be an optimal assignment for $L \cup V \cup H$ such that $ML \subseteq D$. Let $D(H^-)$ be assignment D after eliminating the set of nets H. Let $D(MV)$ be assignment D after routing all nets in $L \cup V$ as in MV. In what follows we use the notation $LB(f, s)$ to denote a lower bound for function f computed using only the set of nets s. Let w represent the width of the rectangles and h represent the sum of the height of each rectangle.

From Theorem 2, we know that

$$\left(1 + \frac{\Delta w'}{w'}\right) \cdot \left(1 + \frac{\Delta h'}{h'}\right) = \left(\frac{w' + \Delta w'}{w'}\right) \cdot \left(\frac{h' + \Delta h'}{h'}\right) \leq 2 \ , \text{ where ,}$$

$$\begin{aligned}
\Delta w' &= (H_{MV}(LL) + H_{MV}(RR)) - (H_{D(H^-)}(LL) + H_{D(H^-)}(RR)) \ , \\
w' &= LB(H_D(LL) + H_D(RR) + w, L \cup V) \ , \\
\Delta h' &= H_{MV}(TMB) - H_{D(H^-)}(TMB) \ , \text{ and} \\
h' &= LB(H_D(TMB) + h, L \cup V) \ .
\end{aligned}$$

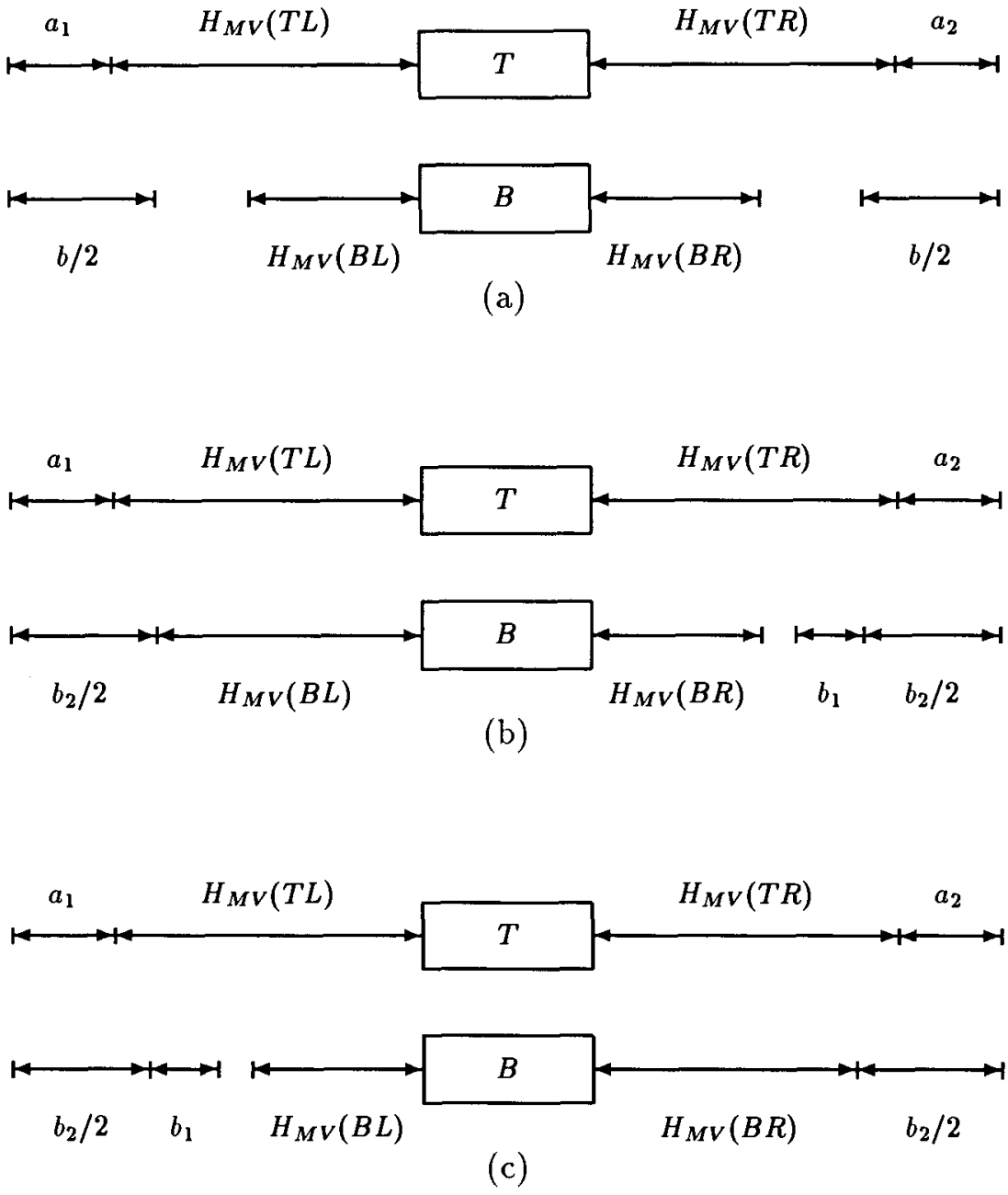


Fig. 9. Examples for case 1.

For the assignment I that corresponds to $D(MV)$, i.e., $H_{D(MV)}(TL) = H_I(TL)$, we know that

$$\left(1+\frac{(H_I(LL)+H_I(RR))-(H_D(LL)+H_D(RR))}{LB(H_D(LL)+H_D(RR)+w, L\cup V\cup H)}\right)\left(1+\frac{H_I(TMB)-H_D(TMB)}{LB(H_D(TMB)+h, L\cup V\cup H)}\right)$$

Replacing the above bound, we know the above equation equals XY, where

$$X = 1 + \frac{\Delta w' + (H_I(LL) + H_I(RR)) - (H_{D(MV)}(LL) + H_{D(MV)}(RR))}{w' + LB(H_D(LL) + H_D(RR) + w, H)} \quad (1)$$

$$Y = 1 + \frac{\Delta h' + H_I(TMB) - H_{D(MV)}(TMB)}{h' + LB(H_D(TMB) + h, H)} \quad (2)$$

Therefore, it is equal to $(1 + \frac{\Delta w' + \Delta w''}{w' + w''})(1 + \frac{\Delta h' + \Delta h''}{h' + h''})$, where

$$\begin{aligned} \Delta w'' &= (H_I(LL) + H_I(RR)) - (H_{D(MV)}(LL) + H_{D(MV)}(RR)) , \\ w'' &= LB(H_D(LL) + H_D(RR) + w, H) , \\ \Delta h'' &= H_I(TMB) - H_{D(MV)}(TMB) , \text{ and} \\ h'' &= LB(H_D(TMB) + h, H) . \end{aligned}$$

Let us now establish Lemma 13 where we prove conditions under which the above approximation bound is two. Later on we show that these conditions hold.

Lemma 13. *Let* $w' \geq q$, $\Delta w' \leq p$, $h' \geq s$, $\Delta h' \leq r$, $w'' \geq x$, $\Delta w'' \leq -y$, $h'' \geq x$ *and* $\Delta h'' \leq x + y$. *Assume that* x, q, $s > 0$; y, $p \geq 0$; $r \leq 0$; $((p+q)(r+s))/qs \leq 2$; $p + q \leq s$; $2p \leq s$ *and* $0 \leq y \leq x$. *Then*

$$\left(1 + \frac{\Delta w' + \Delta w''}{w' + w''}\right)\left(1 + \frac{\Delta h' + \Delta h''}{h' + h''}\right) \leq 2 .$$

Proof. Substituting the bounds for w', $\Delta w'$, h', $\Delta h'$, w'', $\Delta w''$, h'' and $\Delta h''$, we know that

$$\left(1 + \frac{\Delta w' + \Delta w''}{w' + w''}\right) \cdot \left(1 + \frac{\Delta h' + \Delta h''}{h' + h''}\right) \leq \left(1 + \frac{p - y}{q + x}\right) \cdot \left(1 + \frac{r + x + y}{s + x}\right)$$

which is equal to

$$\frac{(p+q)(r+s) + (p+q)(2x+y) + (r+s)(x-y) + (x-y)(2x+y)}{qs + qx + sx + x^2} .$$

Substituting $(p + q)(r + s) \leq 2qs$ and expanding terms, we know the expression is

$$\leq \frac{2qs + 2px + 2qx + (p+q)y + r(x-y) + sx - sy + 2x^2 - xy - y^2}{qs + qx + sx + x^2} .$$

Substituting $p + q \leq s$ and $2p \leq s$, and rearranging terms,

$$\leq \frac{2qs + 2sx + 2qx + r(x-y) + 2x^2 - xy - y^2}{qs + qx + sx + x^2} .$$

Eliminating all the non-positive terms (remember that $r \leq 0$ and $x \geq y$),

$$\leq \frac{2qs + 2sx + 2qx + 2x^2}{qs + qx + sx + x^2} = 2 .$$

This completes the proof of the lemma. □

By definition p, q, $s \geq 0$ and $r \leq 0$ and in what follows we define $x \geq y \geq 0$. If q or s is zero, then it must be that $p = q = r = s = 0$ and the layout constructed in the first stage is optimal. If $x = 0$, then since $x \geq y \geq 0$ we know that the layout constructed in the second stage is optimal. If both of these layouts are optimal, then the approximation bound holds. If only one of the layouts is optimal, then a proof similar to the one in Lemma 13 can be used to show that the approximation bound is at most two. The remaining case is when x, p, q, $s > 0$ and $r \leq 0$. From Theorem 2 we know that $((p + q) \cdot (r + s))/qs \leq 2$. The assumptions that $p + q \leq s$ and $2p \leq s$ can be easily verified in each of the four cases in the proof of Theorem 2.[‡] In the proof of Theorem 3 we define x and y in such a way that $x \geq y \geq 0$. Therefore, all the assumptions in the statement of Lemma 13 hold.

Let D be an optimal assignment for $L \cup V \cup H$ and let $D(MV)$ be as defined before. Let assignment I be one of the preliminary assignments constructed by our algorithm that corresponds to $D(MV)$ (i.e., $H_{D(MV)}(TL) = H_I(TL)$). Let s be the number of wires (from the nets in H) crossing the left side of T and let t be the number of wires (from the nets in H) crossing the right side of T in assignment I. Clearly, in assignment $D(MV)$ there are s wires (from the nets in H) crossing the left side of T and t wires (from the nets in H) crossing the right side of T. We use the function $sp(s,t)$ to denote the maximum increase of $H_{D(MV)}(TMB)$ when we transform assignment MV so that the $s + t$ wires identified above are routed as in assignment I. Clearly, the maximum number of pair of wires that need to be interchanged is $g \leq \min\{s,t\}$. By construction, each of these g pair of wires do not cross on the top side of T in assignment I, but they cross on assignment $D(MV)$. Therefore, each time that we interchange a pair of these wires we increase the vertical height of the assignment by at most two. Hence, the maximum increase of $H_{D(MV)}(TMB)$ is at most $s+t$. We say that when we transform an assignment, $D(MV)$, with s wires that cross the left side of rectangle B (from the nets in H) and t wires that cross the right side of rectangle B (from the nets in H), to assignment I with u wires crossing the left side of rectangle B (from the nets in H) and v wires crossing the right side of rectangle B (from the nets in H), the maximum increase of $H_{D(MV)}(TMB)$ is given by the function

[‡]Note that $p \leq q$ in all cases in the proof of Theorem 2. This together with the fact that $p + q \leq s$ is enough to prove that $2p \leq s$. We did not used this approach because when we reapply these arguments in Theorem 6, the bound $p \leq q$ does not hold.

$$2\max\{\min\{t,u\},\min\{v,s\}\} .$$

The justification for this formula is a straight forward generalization of the previous case.

Theorem 3. *For the $2_C - R2M$ problem, let D be an optimal assignment such that $ML \subseteq D$ and let MH be the assignment generated by our algorithm. Then, $A(MH) \leq 2A(D)$.*

Proof. Let I be the assignment with $H_I(TL) = H_{D(MV)}(TL)$. In what follows we show that $A(I) \leq 2A(D)$. Since $A(MH) \leq A(I)$, we know that $A(MH) \leq 2A(D)$. Assume without loss of generality that $H_{MV}(TL) + H_{MV}(TR) + a \geq H_{MV}(BL) + H_{MV}(BR) + b$, and b is even. The proof for the other cases is similar. Clearly, each net in H is routed in D by a wire that crosses at least two corners. Therefore, $h'' \geq a + b$. This bound together with the fact that each net in H has at least two pins on a horizontal size of T and/or B, we know that $w'' \geq a + b$. Also, since

$$\begin{aligned} H_I(LL) + H_I(RR) = H_I(TL) + H_I(TR) &= H_{D(MV)}(TL) + H_{D(MV)}(TR) \\ &\leq H_{D(MV)}(LL) + H_{D(MV)}(RR) , \end{aligned}$$

we know that $\Delta w'' \leq 0$. In what follows we establish a bound on $\Delta h''$ and, when needed, a sharper bound for $\Delta w''$. There are four cases depending on the values of $H_{MV}(TL)$, $H_{MV}(TR)$, $H_{MV}(BL)$, and $H_{MV}(BR)$.

Case 1: $H_{MV}(LL) = H_{MV}(TL)$ and $H_{MV}(RR) = H_{MV}(TR)$.

There are three subcases depending on the values of a, b, $H_{MV}(TL)$, $H_{MV}(TR)$, $H_{MV}(BL)$, and $H_{MV}(BR)$ (see Fig. 9).

Subcase 1.1: Assignment I is of the form given by Fig. 9(a).

The optimal assignment D after transforming it to $D(MV)$ is given by Fig. 10. From the above discussion we know that transforming assignment $D(MV)$ to assignment I increases $H_{D(MV)}(TMB)$ by at most

$$\Delta h'' \leq sp(a_1, a_2) + 2\max\{\min\{b - b', b/2\}, \min\{b/2, b'\}\} \leq a + b .$$

From above, we know that $\Delta w'' \leq 0$, $w'' \geq a + b$ and $h'' \geq a + b$. Setting $x = a + b$, and $y = 0$, and applying Lemma 13, we know that $A(I) \leq 2A(D)$ for this subcase.

Subcase 1.2: Assignment I is of the form given by Fig. 9(b).

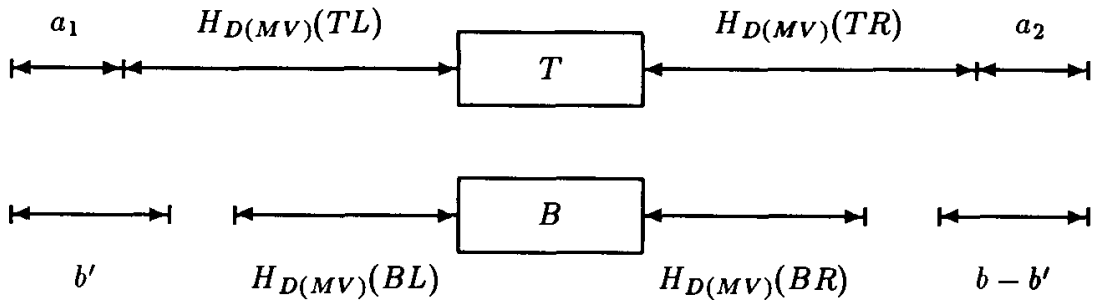


Fig. 10. Assignment $D(MV)$ for subcase 1.1.

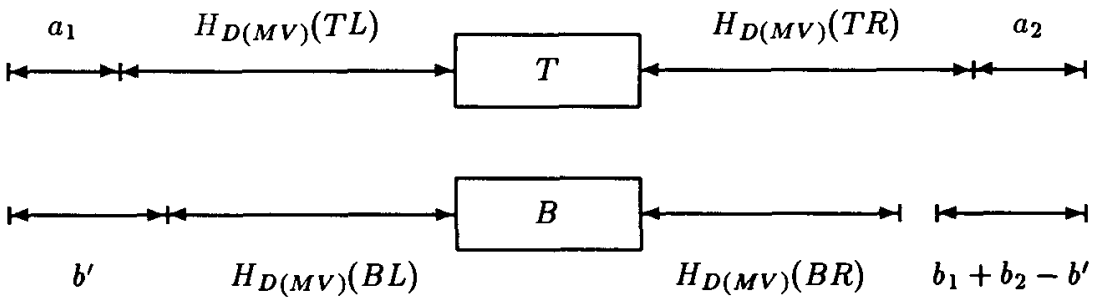


Fig. 11. Assignment $D(MV)$ for subcase 1.2.

The optimal assignment D after transforming it to $D(MV)$ is given by Fig. 11. From the above discussion we know that transforming assignment $D(MV)$ to assignment I increases $H_{D(MV)}(TMB)$ by at most

$$\Delta h'' \leq sp(a_1, a_2) + 2\mathbf{max}\{\mathbf{min}\{b', b_1 + b_2/2\}, \mathbf{min}\{b_1 + b_2 - b', b_2/2\}\} .$$

There are three subcases depending on the value for b'.

Subcase 1.2.1: $b' \leq b_2/2$.

Clearly, $\Delta h'' \leq a + b_2$ and $\Delta w'' \leq 0$. The proof now proceeds as the one for subcase 1.1.

Subcase 1.2.2: $b_2/2 \leq b' \leq b_1 + b_2/2$.

Let $z = b' - b_2/2$. Clearly, $z \leq b_1$, $\Delta h'' \leq a + b_2 + 2z$ and since

$$b' + H_{D(MV)}(BL) - a_1 - H_{D(MV)}(TL) = z$$

(remember that $b_2/2 + H_I(BL) = a_1 + H_I(TL)$), we know that $\Delta w'' \leq -z$. From above we know that $w'' \geq a + b \geq a + b_2 + z$, and $h'' \geq a + b \geq a + b_2 + z$. Setting $x = a + b_2 + z$, and $y = z$, and applying Lemma 13, we know that $A(I) \leq 2A(D)$ for this subcase.

Subcase 1.2.3: $b' \geq b_1 + b_2/2$.

Clearly $\Delta h'' \leq a + b_2 + 2b_1$ and since $b' + H_{D(MV)}(BL) - a_1 - H_{D(MV)}(TL) \geq b_1$, we know that $\Delta w'' \leq -b_1$. From above we know that $w'' \geq a + b \geq a + b_2 + b_1$, and $h'' \geq a + b \geq a + b_2 + b_1$. Setting $x = a + b_2 + b_1$, and $y = b_1$, and applying Lemma 13, we know that $A(I) \leq 2A(D)$ for this subcase.

Subcase 1.3: Assignment I is of the form given by Fig. 9(c).

The proof for this case is omitted since it is similar to the one for subcase 1.2.

The proof for remaining cases is omitted since it is similar to the one for case 1. This completes the proof of the lemma. □

The main problem with the above algorithm is that it is unlikely that it can be implemented to take $O(n)$ time. The reason is that one could generate $\Omega(n)$ layouts each requiring $\Omega(n)$ time. However, an $O(n)$ algorithm that generates solutions within a factor of two of the optimal solution exists. The problem with this new algorithm is that the proof that it generates solutions within two of optimal involves many cases. For brevity we do not include the algorithm nor its proof, however, we discuss the basic idea behind it. An interested reader can derive the algorithm and the proof for the approximation bound from our description. Let us assume that $H_{MV}(TL) + H_{MV}(TR) + a \geq H_{MV}(BL) + H_{MV}(BR) + b$ and that b is even. We also assume that "a" is even (note that when this is not the case, more layouts need to be constructed). Instead of generating "a" layouts, we only generate a constant number of layouts and then select the best of these layouts as our solution. First we set $tl = tr = a/2$. Let us suppose case 1 in procedure FIND holds. We use case 1 of procedure FIND to generate one of the three layouts (see Figs. 9(a)–(c)). If we generate the layout in Fig. 8(a), no other layout needs to be constructed. Let us now consider the case in Fig. 9(b) (the case in Fig. 9(c) is treated similarly to the case in Fig. 9(b)). In this case we generate another layout (see Fig. 12).

The proof for the approximation bound of two is similar to the one in Theorem 3. The main difference is that we need to compare the optimal area layout against one of the two layouts generated. In the proof of Theorem 3 when b' is small, we compare it against the assignment given in Fig. 12(a), but when b' is large it gets compared against the assignments given in Figs. 12(b) or (c). We state the following theorem without proving it.

Theorem 4. *For the $2_C - R2M$ problem, let D be an optimal assignment such that $ML \subseteq D$ and let MH be the assignment generated by our algorithm that generates only a constant number of intermediate rectangles. Then, $A(MH) \leq 2A(D)$.*

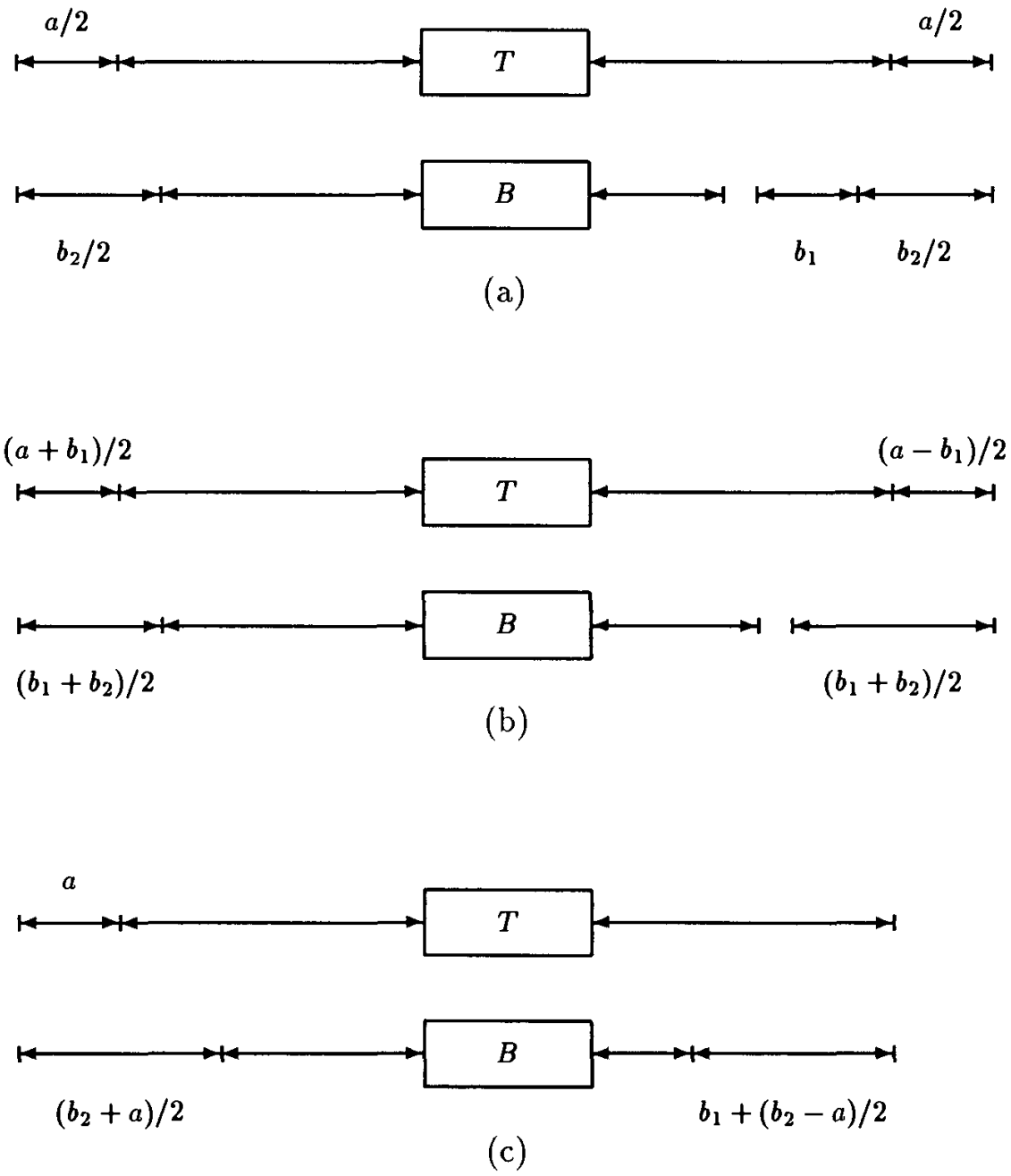


Fig. 12. (a) Assignment Fig. 9(b), (b) assignment constructed when $a \geq b_1$, and (c) assignment constructed when $a < b_1$.

5. Approximation Algorithm for the $2 - R2M$ Problem

In this section we show that our algorithm takes $O(n)$ time and generates a layout with area at most $2\,OPT$, where OPT is the area of an optimal layout.

algorithm for the $2-R2M$ problem
  Construct assignments ML, MS, and MH
    (the one just before the proof of Theorem 4);
  Construct and output a layout with area $A(MH)$ for MH.
end of algorithm

Theorem 5. *The time complexity of our algorithm is* $O(n)$.

Proof. The assignment constructed in Sec. 4.1 can be obtained in $O(n)$ time by a simple recursive procedure that manipulates two priority queues and uses a simple marking scheme.[12] It is simple to verify that the remaining part of the assignment and the final layout can be constructed in $O(n)$ time. □

Theorem 6. *Let* D *be an optimal assignment such that* $ML \subseteq D$ *and let* Q *be the layout generated by our algorithm. Then,* $A(Q) \leq 2A(D)$.

Proof. Arguments similar to those used in Theorems 1 and 4 can be used to prove this theorem. The main difference is that one needs to justify the assumptions in Lemma 13 for this more general case (i.e., arguments similar to those that follow Lemma 13). □

6. Discussion

We have presented an efficient approximation algorithm that generates a layout with area within a factor of two of the area in an optimal layout for the $2-R2M$ problem. The algorithm takes $O(n \log n)$ ($O(n)$ if the set of terminals is initially sorted) time and the constant associated with this bound is small. It is possible to obtain a better solution (not a necessarily a better approximation bound) by using the optimal algorithms for $R1M$ problem to route the set of nets in C. Our results also hold when the rectangles are placed side by side, and both rectangles have equal height. For brevity this other problem is not discussed in detail. The worst case scenario for our approximation algorithm does not arise all of the time. We suspect that most of the time our solutions are near optimal. When implementing the algorithm it is simple to compute our lower bound for the area of an optimal area layout. This gives a good estimate of how close from optimal is the assignment generated by the algorithm.

In Sec. 1 we made the middle-channel assumption. We can remove this assumption at the expense of an additional layer when under the knock-knee wiring model. In this case the routing in the middle channel is performed by a well known algorithm.[3] Before one can approximate efficiently the two-layer Manhattan mode $2-R2M$ problem, we need an efficient approximation algorithm for approximating the area of the channel routing problem under the Manhattan model. It is not clear whether or not such algorithm exits.

With respect to the multiterminal net case ($R2M$ problem) we do not have an efficient approximation algorithm for it. The main problem is finding an approximation algorithm to approximate the area of the channel routing problem. A good

heuristic for this case can be based on Gonzalez and Lee's algorithm.[11] Another intersting open problem is when the rectangles have different widths and can slide horizontally.

Acknowledgements

This research was supported in part by the National Science Foundation under Grant DCR-8503163, and by the National Science Council, Taiwan, R.O.C., under grant NCS 81-0404-E194-502. A preliminary version of this paper appears in the Proceedings of the Twenty-Fourth Annual Allerton Conference on Communication, Control and Computing, Oct. 1986, pp. 550-559.

References

1. C. K. Kim, "GRADER: A fast heuristic global router", *Proceedings of the 1987 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, 341–345.
2. M. Sarrafzadeh and F. P. Preparata "A bottom-up layout technique based on two-rectangle routing", *INTEGRATION: The VLSI Journal*, **5** (1987) 231–246.
3. R. Kuchem, D. Wagner and F. Wagner, "Area optimal three layer channel routing", *Proceedings of the 1989 Foundations of Computer Science Conference*, 506–511.
4. T. Szymanski and M. Yannakakis, Personal Communication,[16] (1982).
5. M. Sarrafzadeh, "Channel-routing problem in the knock-knee mode is NP-complete," *IEEE Transactions on Computer Aided Design*, **CAD-6(4)** (1987) 293–303.
6. A. Hashimoto and J. E. Stevens, "Wire routing by optimizing channel assignment without large apertures", *Proceedings of the 8th IEEE Design Automation Conference*, (1971), 155–169.
7. U. I. Gupta, D. T. Lee and J. Leung, "An optimal solution for the channel assignment problem", *IEEE Transactions on Computers*, **C-28(11)** (1979) 907–810.
8. A. S. LaPaugh, "A polynomial time algorithm for optimal routing around a rectangle", *Proceedings of the 21st IEEE Foundation of Computer Science*, (1980) 282–293.
9. A. S. LaPaugh, "Algorithms for integrated circuit layout, an analytic approach", Ph.D. dissertation, Massachusetts Institute of Technology, (1980).
10. T. Gonzalez and S. Lee, "A linear time algorithm for optimal wiring around a rectangle", *Journal of the ACM*, **35(4)** (1988) 810–832.
11. T. Gonzalez and S. Lee, "Routing multiterminal nets around a rectangle", *IEEE Transactions on Computers*. **C-35(6)** (1986) 543–549.
12. T. Gonzalez and S. Lee, "A 1.6 approximation algorithm for routing multiterminal nets", *SIAM J. Computing*, **16(4)** (1987) 669–704.
13. B. Sahni, A. Bhatt and R. Raghavan, "The complexity of design automation problems", *Proceedings of the 19th Design Automation Conference*, (1981).
14. M. S. Chandrasekhar and M. A. Breuer, "Optimum placement of two rectangular blocks", *Proceedings of the 19th Design Automation Conference*, (1982) 879–886.
15. B. S. Baker, "A provable good algorithm for the two module routing problem", *SIAM J. Computing*, **15(1)** (1986) 162–187.
16. B. S. Baker, S. N. Bhatt and F. T. Leighton, "An approximation algorithm for Manhattan routing", *Advances in Computer Research*, ed. F. P. Preparata, **2** (1984) 205–229.
17. R. Pinter, "Optimal routing in rectilinear channels", *VLSI Systems and Computations*, Computer Science Press, Eds. H. T. Kung, R. Sproull and G. Steele, (1981), 160–177.

Appendix A

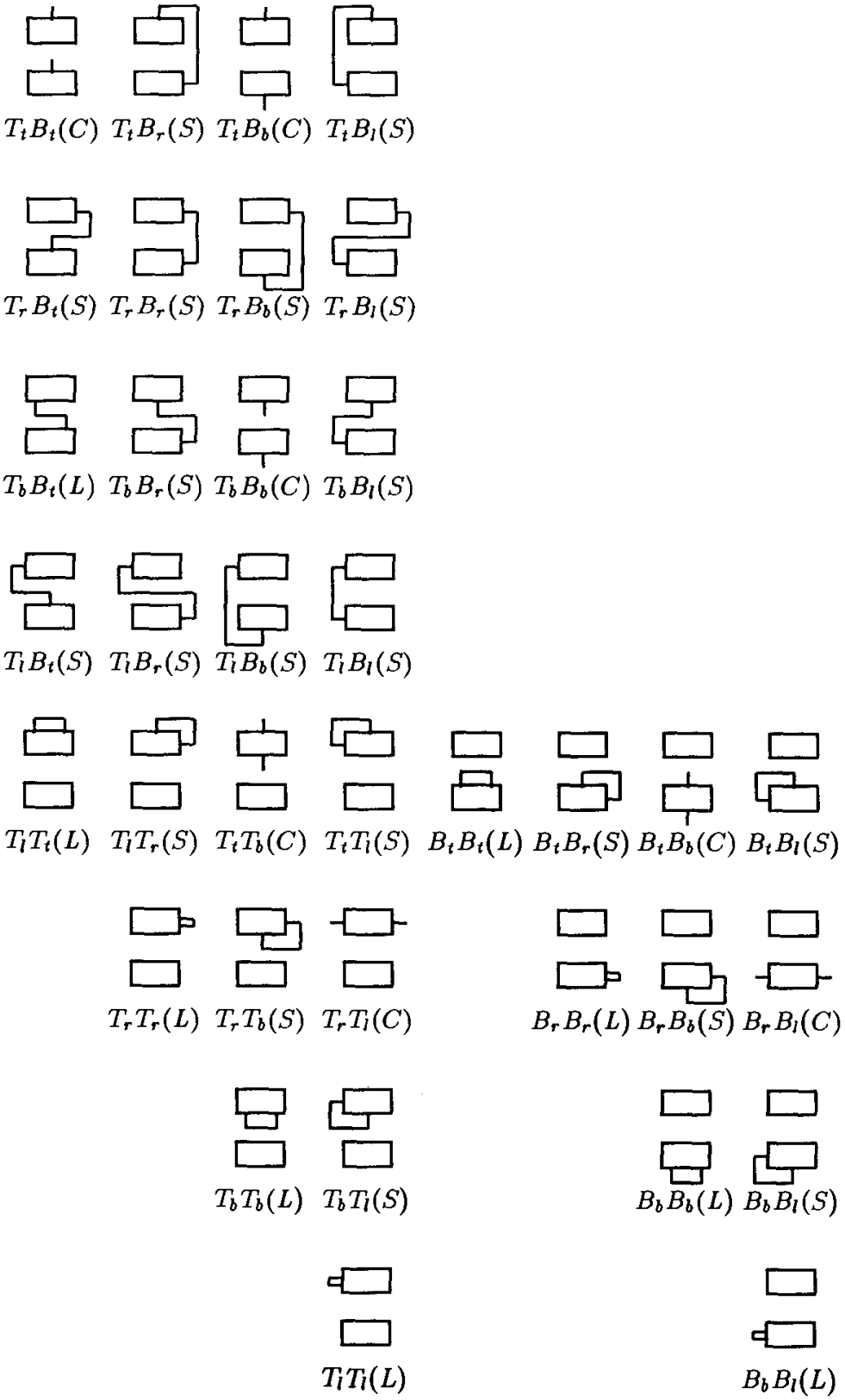

$T_tB_t(C)$ $T_tB_r(S)$ $T_tB_b(C)$ $T_tB_l(S)$
$T_rB_t(S)$ $T_rB_r(S)$ $T_rB_b(S)$ $T_rB_l(S)$
$T_bB_t(L)$ $T_bB_r(S)$ $T_bB_b(C)$ $T_bB_l(S)$
$T_lB_t(S)$ $T_lB_r(S)$ $T_lB_b(S)$ $T_lB_l(S)$
$T_lT_t(L)$ $T_lT_r(S)$ $T_tT_b(C)$ $T_tT_l(S)$ $B_tB_t(L)$ $B_tB_r(S)$ $B_tB_b(C)$ $B_tB_l(S)$
$T_rT_r(L)$ $T_rT_b(S)$ $T_rT_l(C)$ $B_rB_r(L)$ $B_rB_b(S)$ $B_rB_l(C)$
$T_bT_b(L)$ $T_bT_l(S)$ $B_bB_b(L)$ $B_bB_l(S)$
$T_lT_l(L)$ $B_bB_l(L)$

Fig. A.1. Types of nets.

M. Sarrafzadeh
D. T. Lee

ALGORITHMIC ASPECTS OF VLSI LAYOUT